

一、C++基础知识

1. 程序的注释

1. 单行注释使用符号` `//` `
2. 多行注释使用符号` `/*` `

2. 常量

1. ``#define 常量名 常量值``
2. ``const 数据类型 常量名 = 常量值``

1.4 常量

作用：用于记录程序中不可更改的数据

C++定义常量两种方式

1. #define 宏常量： `#define 常量名 常量值`
 - 通常在文件上方定义，表示一个常量
2. const修饰的变量 `const 数据类型 常量名 = 常量值`
 - 通常在变量定义前加关键字const，修饰该变量为常量，不可修改

3. 标识符的命名规则

1.6 标识符命名规则

作用：C++规定给标识符（变量、常量）命名时，有一套自己的规则

- 标识符不能是关键字
- 标识符只能由字母、数字、下划线组成
- 第一个字符必须为字母或下划线
- 标识符中字母区分大小写

建议：给标识符命名时，争取做到见名知意的效果，方便自己和他人的阅读

二、数据类型

1. 整型

作用：整型变量表示的是整数类型的数据

C++中能够表示整型的类型有以下几种方式，区别在于所占内存空间不同：

数据类型	占用空间	取值范围
short(短整型)	2字节	(-2^15 ~ 2^15-1)
int(整型)	4字节	(-2^31 ~ 2^31-1)
long(长整形)	Windows为4字节，Linux为4字节(32位)，8字节(64位)	(-2^31 ~ 2^31-1)
long long(长长整形)	8字节	(-2^63 ~ 2^63-1)

2. sizeof关键字

统计所有类型所占内存大小

3. 实型（浮点型）

1. 单精度float
 2. 双精度double

数据类型	占用空间	有效位数
float	4	7位有效数字
double	8	15-16位有效数字

4. 字符型

1. 作用：字符型变量用于显示单个字符
2. 语法：char ch = 'a';

5. 转义字符

常用的有：`\n \\\t`

转义字符	含义	ASCII码值(十进制)
\a	警报	007
\b	退格(BS)，将当前位置移到前一列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF)，将当前位置移到下一行开头	010
\r	回车(CR)，将当前位置移到本行开头	013
\t	水平制表(HT) (跳到下一个TAB位置)	009
\v	垂直制表(VT)	011
\\\	代表一个反斜线字符"\\"	092
\'	代表一个单引号(撇号)字符	039
\"	代表一个双引号字符	034
\?	代表一个问号	063



6. 字符串

c风格字符串：char 变量名[] = "字符串值"

c++风格字符串：string 变量名 = "字符串值"

7. 布尔类型值bool

8. 数据的输入

1. 关键字：cin
2. 语法：cin >> 变量

运算符

1. 算数运算符

运算符	术语	示例	结果
+	正号	+3	3
-	负号	-3	-3
+	加	10 + 5	15
-	减	10 - 5	5
*	乘	10 * 5	50
/	除	10 / 5	2
%	取模(取余)	10 % 3	1
++	前置递增	a=2; b=++a;	a=3; b=3;
++	后置递增	a=2; b=a++;	a=3; b=2;
--	前置递减	a=2; b=--a;	a=1; b=1;
--	后置递减	a=2; b=a--;	a=1; b=2;

前置递增， ++a先加1再参与表达式运算 后置递增， a++先参与表达式计算再加1

2. 逻辑运算符

作用：用于根据表达式的值返回真值或假值

逻辑运算符有以下符号：

运算符	术语	示例	结果
!	非	!a	如果a为假，则!a为真； 如果a为真，则!a为假。
&&	与	a && b	如果a和b都为真，则结果为真，否则为假。
	或	a b	如果a和b有一个为真，则结果为真，二者都为假时，结果为假。

三、程序流程结构

1. 选择结构

1. 三目运算符：语法：``表达式1 ? 表达式2 : 表达式3``

表达式1为真运行表达式2

表达式1为假运行表达式3

2. switch语句：表达式类型只能是整形或者字符型

2. 循环结构

3. 跳转语句

1. goto语句:

语法: goto 标记

四、数组

1. 一维数组

一维数组名的用途：

一维数组名称的用途：

1. 可以统计整个数组在内存中的长度
2. 可以获取数组在内存中的首地址

2. 冒泡排序

作用：最常用的排序算法，对数组内元素进行排序

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素做同样的工作，执行完毕后，找到第一个最大值。
3. 重复以上的步骤，每次比较次数-1，直到不需要比较



示例：将数组 { 4,2,8,0,5,7,1,3,9 } 进行升序排序

```
1 int main() {
2
3     int arr[9] = { 4,2,8,0,5,7,1,3,9 };
4
5     for (int i = 0; i < 9 - 1; i++)
6     {
7         for (int j = 0; j < 9 - 1 - i; j++)
8         {
9             if (arr[j] > arr[j + 1])
10            {
11                 int temp = arr[j];
```

五、函数

1. 定义

函数的定义一般主要有5个步骤：

- 1、返回值类型
- 2、函数名
- 3、参数表列
- 4、函数体语句
- 5、return 表达式

2. 语法

```
返回值类型 函数名 (参数列表)
{
    函数体语句
    return 表达式
}
```

3. 函数的声明

作用：告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

- 函数的声明可以多次，但是函数的定义只能有一次

4. 函数的分文件编写

作用：让代码结构更加清晰

函数分文件编写一般有4个步骤

1. 创建后缀名为.h的头文件
2. 创建后缀名为.cpp的源文件
3. 在头文件中写函数的声明
4. 在源文件中写函数的定义

六、指针

1. 基本概念

1. 作用：间接访问内存
2. 内存编号从0开始记录，一般为16进制表示
3. 可以利用指针变量保存地址

语法：数据类型 * 变量名；

2. 空指针和野指针

1. 空指针：

1. 用于给指针变量初始化
 2. 空指针是不可以进行访问的
2. 野指针：指针变量指向非法的内存空间

3. 常量指针和指针常量

```
const int * p = &a;  
常量指针  
特点：指针的指向可以修改，但  
是指针指向的值不可以改
```

*p = 20; 错误，指针指向的值不可以改
p = &b; 正确，指针指向可以改

1. 常量指针

```
int * const p = &a;
```

指针常量

特点：指针的指向不可以改，指针指向的值可以改

*p = 20; 正确，指向的值可以改

p = &b; 错误，指针指向不可以改

2. 指针常量

七 结构体

1. 结构体是用户自定义的数据类型，允许用户存储不同的数据类型

2. 结构体的定义和使用：

1. 语法：`struct 结构体名 {结构体成员列表};`

2.

通过结构体创建变量的方式有三种：

- `struct 结构体名 变量名`
- `struct 结构体名 变量名 = { 成员1值 , 成员2值...}`
- 定义结构体时顺便创建变量

总结1：定义结构体时的关键字是`struct`，不可省略

总结2：创建结构体变量时，关键字`struct`可以省略

总结3：结构体变量利用操作符`."`访问成员

3. 结构体数组

1. 作用：将自定义的结构体放入到数组中方便维护

2. 语法：`struct 结构体名 数组名[元素个数] = {{}, {}, {}};`

4. 结构体指针

作用：通过指针访问结构体中的成员

- 利用操作符 `->` 可以通过结构体指针访问结构体属性

C++核心编程

1. 内存分模型

- 代码区：存放函数的二进制代码，由操作系统进行管理
- 全局区：存放全局变量和静态变量以及常量
- 栈区：由编译器自动分配释放，存放函数的参数值、局部变量等
- 堆区：由程序员进行分配和释放，若程序员不释放，程序结束时由操作系统回收

1.1 程序运行前

在程序编译后，生成了exe可执行程序，未执行该程序前分为两个区域

代码区：

存放 CPU 执行的机器指令

代码区是共享的，共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可

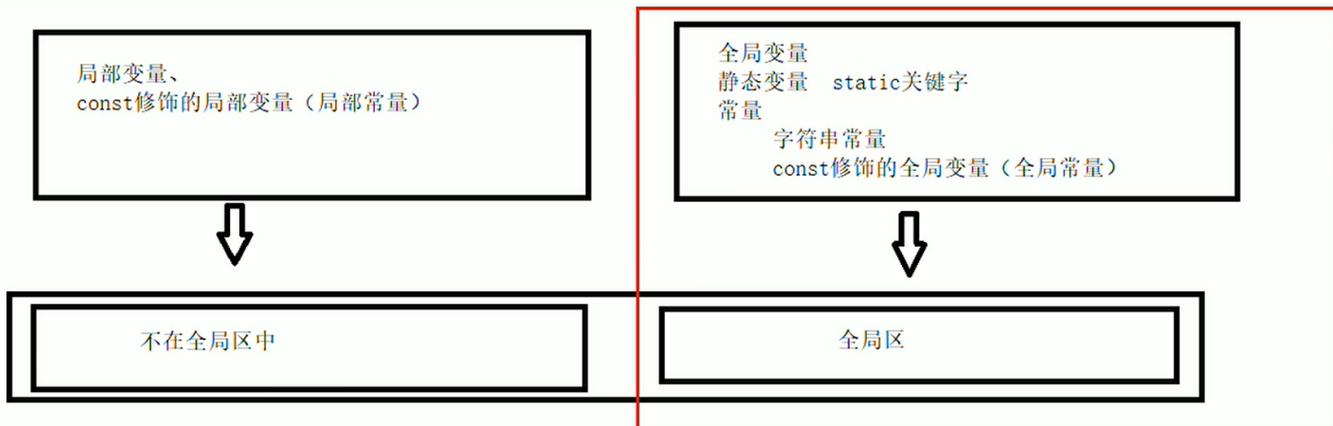
代码区是只读的，使其只读的原因是防止程序意外地修改了它的指令

全局区：

全局变量和静态变量存放在此。

全局区还包含了常量区，字符串常量和其他常量也存放在此。

该区域的数据在程序结束后由操作系统释放。



- C++在程序运行前分为全局区和代码区
- 代码区特点是只读和共享
- 全局区中存放全局变量、静态变量和常量
- 常量区中存放 const修饰的全局常量 和 字符串常量

1.2 程序运行后

栈区：

由编译器自动分配释放, 存放函数的参数值, 局部变量等

注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

堆区：

由程序员分配释放, 若程序员不释放, 程序结束时由操作系统回收

在C++中主要利用new在堆区开辟内存

1.3 new操作符

C++中利用new操作符在堆区开辟数据

语法： new 数据类型

利用new创建的数据，会返回数据对应的类型的指针

释放堆区数据用delete，数据类型为数组时用delete[]

2. 引用

2.1 引用的基本使用：

作用：给变量起别名

语法：数据类型 &别名 = 原名

2.2 注意事项

引用必须初始化

引用在初始化后不可改变

2.3 引用做函数参数

作用：函数传参时，可以利用引用的技术让形参修饰实参

优点：可以简化指针修改实参

2.4 引用做函数返回值

- 作用：引用可以作为函数的返回值存在
- 注意：不要返回局部变量的引用
- 用法：函数调用作为左值

2.5 引用的本质

引用的本质在C++内部是一个指针常量

例如：`int& ref = &a`

//自动转换为`int* const ref = &a`，指针常量是指针指向不可以改，也说明为什么引用不可更改

2.6 常量引用

- 作用：常量引用主要用来修饰形参，防止误操作
- 在函数参数列表中，可以加`const`修饰形参，防止形参改变实参

3. 函数提高

3.1 函数默认参数

注意事项：

- 如果某个位置已经有默认参数，这个位置往后从左到右都必须有默认值
- 如果函数声明有默认参数，函数实现就不能有默认参数（声明和实现只能有一个有默认参数）

3.2 函数占位参数

语法：返回值类型 函数名 (数据类型) {}

占位也可以有默认参数

3.3 函数重载

3.3.1 概述

函数重载的满足条件：

- 同一个作用域
- 函数名称相同
- 函数类型不同，或者个数不同，或者顺序不同

注意事项：

- 函数的返回值不可以作为函数重载的条件

例如：

```
void func(){}
int func(){}
```

- 引用作为重载函数

- 函数重载碰到默认参数

4 类和对象

C++面向对象的三大特性为：封装、继承和多态

C++认为万事万物皆为对象，对象上有其属性和行为

4.1 封装

4.1.1 封装的意义

- 将属性和行为作为一个整体，表现生活中的事物
- 将属性和行为加以权限控制

语法：class 类名{ 访问权限： 属性 / 行为 };

```
class Circle
{
    //访问权限
public:
    //属性(半径)
    int m_r;
```

```

//行为
double caculateZC()
{
    return 2 * PI * m_r;
}

};

int main() {

    //创建圆类的对象，实例化
    Circle c1;

    //属性
    c1.m_r = 10;

    //行为
    double ZC = c1.caculateZC();
    cout << "圆的周长为：" << ZC << endl;

    system("pause");
    return 0;
}

```

封装的意义二：

- 类在设计时可以把属性和行为放在不同的权限下，加以控制

访问的权限有三种：

访问权限	解释	特点	区别
public	公共权限	类内可以访问，类外也可以访问	
protected	保护权限	类内可以访问，类外不可以访问	子类可以访问父类中的保护内容
private	私有权限	类内可以访问，类外不可以访问	子类不可以访问父类中的保护内容

4.1.2 struct和class的区别

默认的访问权限不同

- struct的默认访问权限为共有
- class的默认权限为私有

4.1.3 成员属性设置为私有

优点1：将所有的成员属性设置为私有，可以自己控制读写权限

优点2：对于写权限，可以检测数据的有效性

4.2 对象的初始化和清理

4.2.1 构造函数和析构函数

对象的初始化和清理

- 构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无需手动调用
- 析构函数：主要作用在于对象销毁前系统自己调用，执行一些清理工作

构造函数语法：**类名()**{}

1. 构造函数，没有返回值也不写void
2. 函数名称与类名相同
3. 构造函数可以有参数，因此可以发生重载
4. 程序在调用对象的时候会自动调用构造，无需手动调用，而且只会调用一次

析构函数语法：**~类名()**{}

1. 析构函数，没有返回值也不写void
2. 函数名称与类名相同，在名称前加上符号~
3. 析构函数不可以有参数，因此不可以发生重载
4. 程序在调用对象的时候会自动调用析构，无需手动调用，而且只会调用一次

4.2.2 构造函数的分类及调用

两种分类方式：

- 按参数分为：有参构造和无参构造
- 按类型分为：普通构造和拷贝构造

三种调用方式：

- 括号法
- 显示法
- 隐式转换法

```
#include<iostream>;
using namespace std;

//分类
//按照参数分类 无参构造（默认构造）和有参构造
//按照类型分类 普通构造 拷贝构造
class Person
{
public:
    //构造函数
    Person()
    {
        cout << "person 的无参构造函数调用" << endl;
    }
    Person(int a)
    {
        age = a;
        cout << "person 的有参构造函数调用" << endl;
    }
}
```

```
//拷贝构造函数
Person(const Person &p)
{
    //将传入人身上的属性，拷贝到当前对象身上
    age = p.age;
    cout << "person 的拷贝构造函数调用" << endl;
}

//析构函数
~Person()
{
    cout << "person 的析构函数调用" << endl;
}

int age;
};

void test()
{
    //1.括号法
    Person p1;//默认构造法
    Person p2(18);//有参构造法
    Person p3(p2);//拷贝造法

    //注意事项1
    //Person p1();程序会认为是函数声明，不会认为在创建对象

    //2.显示法
    Person p1;
    Person p2 = Person(18);//有参构造
    Person p3 = Person(p2);//拷贝构造

    //Person(10);匿名对象，当前执行结束后，系统会立即回收掉匿名对象

    //注意事项2
    //不要用拷贝构造函数初始化匿名对象，编译器会认为Person (p3) == Person p3;(对象声明)

    //3.隐式括号法
    Person p4 = 10;//有参构造 相当于写了Person p4 = Person(10);
    Person p5 = p4;//拷贝构造
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

4.2.3 拷贝构造函数的调用时机

- 使用一个已经创建完毕的对象来初始化一个新对象
- 值传递的方式给函数参数传值
- 以值方式返回局部变量

```
#include<iostream>
using namespace std;

class Person
{
public:
    Person()
    {
        cout << "Person的默认构造函数调用" << endl;
    }
    Person(int age)
    {
        m_age = age;
        cout << "Person的有参构造函数调用" << endl;
    }
    Person(const Person &p)
    {
        m_age = p.m_age;
        cout << "Person的拷贝构造函数调用" << endl;
    }

    ~Person()
    {
        cout << "Person的析构函数调用" << endl;
    }

    int m_age;
};

//已创建完毕的对象初始化另一个对象
void test01()
{
    Person p1;
    Person p2(p1);
}

//值传递的方式给函数参数传值
void dowork(Person p)
{
}

void test02()
{
    Person p;
    dowork(p);
}
```

```
//以值方式返回局部变量(这里未调用拷贝函数的原因：返回值优化)

Person dowork2()
{
    Person p1;
    cout << "p1的地址为：" << (int*)&p1 << endl;
    return p1;
}
void test03()
{
    Person p = dowork2();
    cout << "p的地址为：" << (int*)&p << endl;
}

int main()
{
    //test01();
    //test02();
    test03();

    system("pause");
    return 0;
}
```

4.2.4 构造函数调用规则

默认情况下，C++编译器至少给一个类添加3个函数

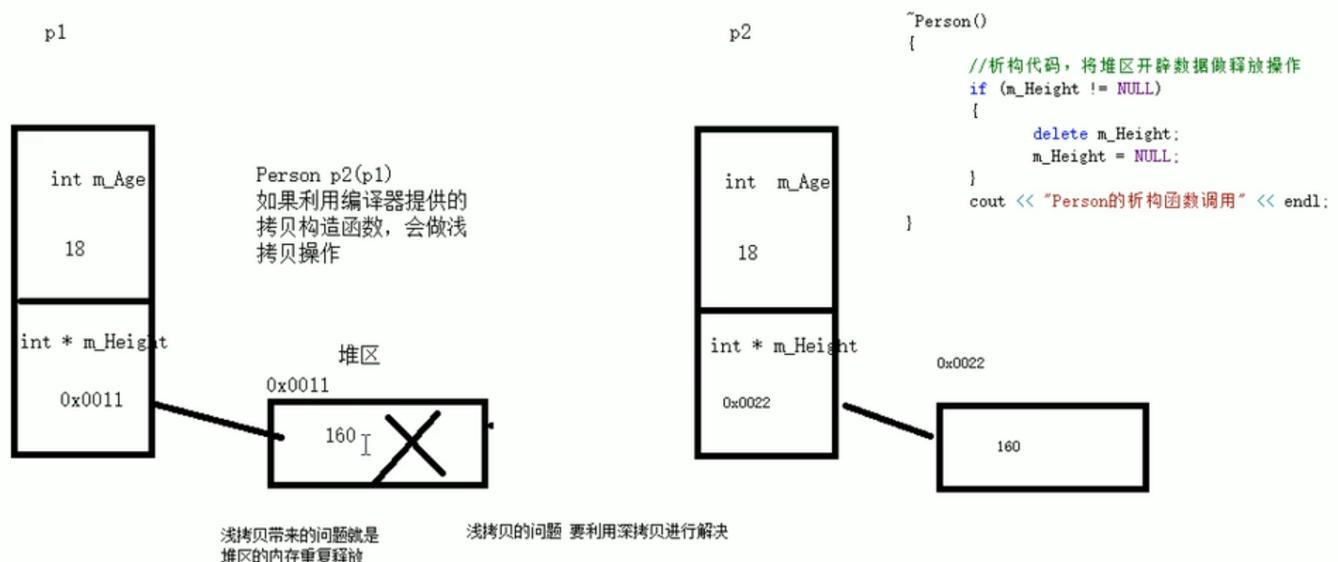
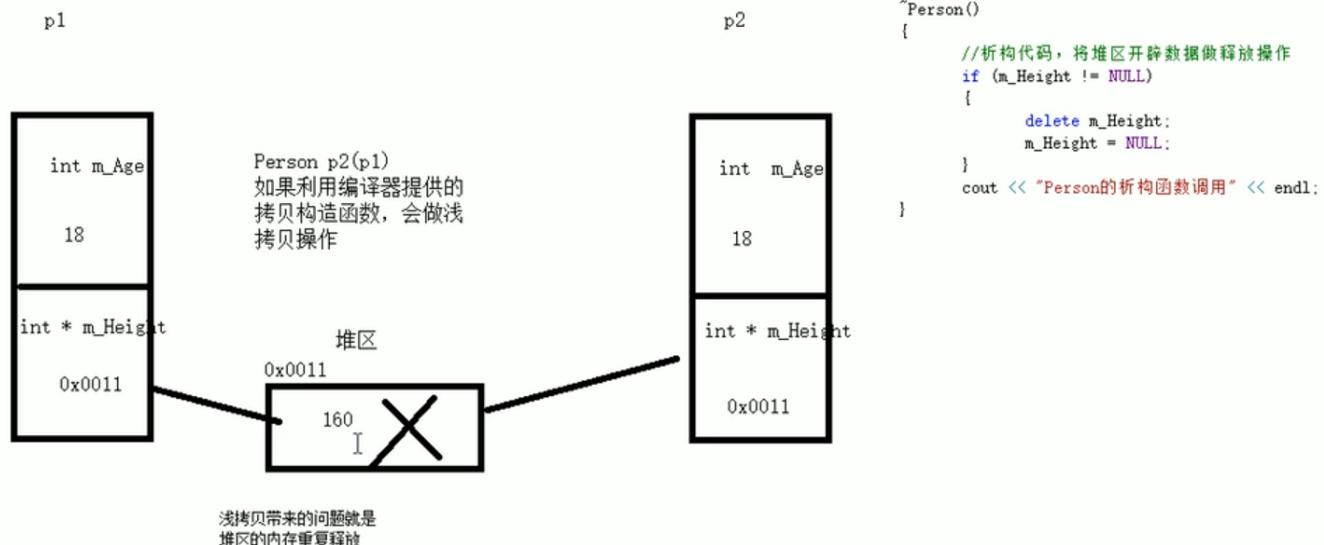
1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝

调用规则如下：

- 如果用户定义有参构造函数、C++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数、C++不会再提供其他构造函数

4.2.5 深拷贝与浅拷贝

- 浅拷贝：简单的赋值操作
- 深拷贝：在堆区重新申请空间，进行拷贝工作



```
#include<iostream>;
using namespace std;

class Person
{
public:
    Person()
    {
        cout << "Person的默认构造函数调用" << endl;
    }

    Person(int age,int height)
    {
        m_age = age;
        m_height = new int(height);
        cout << "Person的有参构造函数调用" << endl;
    }

    Person(const Person& p)
    {
```

```

    m_age = p.m_age;
    m_height = new int(*p.m_height);
    cout << "Person的拷贝构造函数调用" << endl;
}

~Person()
{
    if (m_height != NULL)
    {
        delete m_height;
        m_height = NULL;
    }
    cout << "Person的析构函数调用" << endl;
}

int m_age;
int* m_height;
};

void test01()
{
    Person p1(18,175);
    cout << "p1年龄为: " << p1.m_age << endl;
    cout << "p1身高为: " << *p1.m_height << endl;
    Person p2(p1);
    cout << "p2年龄为: " << p2.m_age << endl;
    cout << "p2身高为: " << *p2.m_height << endl;
}

int main()
{
    test01();

    system("pause");
    return 0;
}

```

4.2.6 初始化列表

C++提供了初始化列表语法，用来初始化属性

语法：构造函数():属性值1(值1),属性值2(值2)...{}

4.2.7 类对象作为类成员

C++类中的成员可以是另一个类的对象，我们称该成员为对象成员

例如

```

class A{}

class B

```

```
{  
    A a;  
}
```

//当类中成员是其他类对象时，我们称该成员为 对象成员

//构造的顺序是：先调用对象成员的构造，再调用本类构造

//析构顺序与构造相反

4.2.8 静态成员

静态成员就是在成员变量和成员函数前加上关键字静态、称为静态成员

- 静态成员变量
 - 所有对象共享同一份数据
 - 在编译阶段分配内存
 - 类内声明，类外初始化
- 静态成员函数
 - 所有对象共享同一个函数
 - 静态成员函数只能访问静态成员变量

静态成员变量两种访问方式：

- 通过对象
- 通过类名

```
//静态成员变量两种访问方式

//1、通过对象
Person p1;
p1.m_A = 100;
cout << "p1.m_A = " << p1.m_A << endl;

Person p2;
p2.m_A = 200;
cout << "p1.m_A = " << p1.m_A << endl; //共享同一份数据
cout << "p2.m_A = " << p2.m_A << endl;

//2、通过类名
cout << "m_A = " << Person::m_A << endl;

//cout << "m_B = " << Person::m_B << endl; //私有权限访问不到
```

静态成员函数两种访问方式：

- 通过对象
- 通过类名

4.3 C++对象模型和this指针

4.3.1 成员变量和成员函数分开存储

在C++中，类内的成员变量和成员函数分开存储

只有非静态成员变量才属于类的对象上

4.3.2 this指针概念

this指针用途：（this指针的本质是指针常量，指针的指向不可以修改）

- 当形参和成员变量同名时，可用this指针来区分
- 在类的非静态成员函数中返回对象本身，可使用return *this

通过4.3.1我们知道在C++中成员变量和成员函数是分开存储的

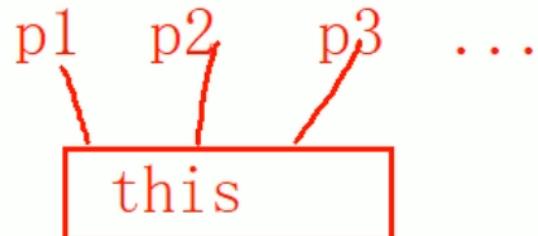
每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码

那么问题是：这一块代码是如何区分那个对象调用自己的呢？

c++通过提供特殊的对象指针，this指针，解决上述问题。**this指针指向被调用的成员函数所属的对象**

this指针是隐含每一个非静态成员函数内的一种指针

this指针不需要定义，直接使用即可



this指针的用途：

- 当形参和成员变量同名时，可用this指针来区分
- 在类的非静态成员函数中返回对象本身，可使用return *this

```
class Person
{
public:

    Person(int age)
    {
        //1、当形参和成员变量同名时，可用this指针来区分
        this->age = age;
    }
}
```

```
Person& PersonAddPerson(Person p)
```

```
{
    this->age += p.age;
    //返回对象本身
    return *this;
}
```

```
int age;
```

```
};
```

```
void test01()
{
    Person p1(10);
    cout << "p1.age = " << p1.age << endl;

    Person p2(10);
    p2.PersonAddPerson(p1).PersonAddPerson(p1).PersonAddPerson(p1);
    cout << "p2.age = " << p2.age << endl;
}

int main() {
    test01();

    system("pause");

    return 0;
}
```

4.3.3 空指针访问成员函数

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到This指针

如果用到This指针，需要加以判断保证代码的健壮性

4.3.4 const修饰成员函数

常函数：

- 成员函数后加const后我们称为这个函数为常函数
- 常函数内不可以修改成员属性
- 成员属性声明时加关键字mutable后，在常函数中依然可以修改

常对象：

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

常函数和常对象的例子：

```
//常函数
class Person
{
public:
    //this指针的本质是指针常量，指针的指向不可以修改
    //const Person* const this
    //在成员函数后面加const，修饰的是This指向，让指针指向的值也不可以修改
    void showperson() const
    {
        this->m_b = 100;
        //this->m_a = 100;
        //this = NULL;      //this指针不可以修改指针指向
    }

    void func()
    {

    }

    int m_a;
    mutable int m_b;    //特殊函数，即使在常函数中也可以修改这个值，加上关键字mutable
};

//常对象
void test02()
{
    const Person p;//在对象前加const，变为常对象
    //p.m_a = 100;
    p.m_b = 100;//m_b是特殊值，在常对象下也可以修改
```

```
//常对象只能调用常函数  
p.showperson();  
//p.func();//常对象不可以调用普通成员函数，因为普通成员函数可以修改属性  
}
```

4.4 友元

友元的关键字：`friend`

- 全局函数做友元
- 类做友元
- 成员函数做友元

1. 全局函数做友元

```
class Person  
{  
    friend void myfriend(Person* p);  
  
public:  
    string sittingroom = "客厅";  
  
private:  
    string bedroom = "卧室";  
};  
  
void myfriend(Person *p)  
{  
    cout << "正在访问：" << p->sittingroom << endl;  
    cout << "正在访问：" << p->bedroom << endl;  
}  
  
void test02()  
{  
    Person p1;  
    myfriend(&p1);  
}
```

2. 类做友元

```
//类做友元  
class Building;  
class Goodfri  
{  
public:  
    Goodfri();
```

```
void visit()//参观函数 用来访问building中的属性

Building* building;

};

class Building
{
    friend class Goodfri;
public:
    Building();
    string sittingroom;

private:
    string bedroom;

};

Building::Building()
{
    sittingroom = "客厅";
    bedroom = "卧室";
}

Goodfri::Goodfri()
{
    building = new Building;

}

void Goodfri::visit()
{
    cout << "好朋友正在访问: " << building->sittingroom << endl;
    cout << "好朋友正在访问: " << building->bedroom << endl;
}

void test01()
{
    Goodfri p1;
    p1.visit();

}
```

3. 成员函数做友元

```
#include<iostream>
using namespace std;

class Building;
class Goodfriend
{
public:
```

```
Goodfriend();

void visit01();//让visit01可以访问building中私有成员
void visit02();//让visit02不可以访问building中私有成员

Building* building;

};

class Building
{
    //告诉编译器 Goodfriend类中的visit01函数 是Building的好朋友，可以访问私有内容
    friend void Goodfriend::visit01();
public:
    Building();
    string sittingroom;

private:
    string bedroom;
};

//类外实现成员函数

Building::Building()
{
    sittingroom = "客厅";
    bedroom = "卧室";
}

Goodfriend::Goodfriend()
{
    building = new Building;
}

void Goodfriend::visit01()
{
    cout << "visit01函数正在访问: " << building->sittingroom << endl;
    cout << "visit01函数正在访问: " << building->bedroom << endl;
}

void Goodfriend::visit02()
{
    cout << "visit02函数正在访问: " << building->sittingroom << endl;
    //cout << "visit函数正在访问: " << building->bedroom << endl;
}

void test01()
{
    Goodfriend p1;
    p1.visit01();
    p1.visit02();

}
```

```
int main()
{
    test01();

    system("pause");
    return 0;
}
```

4.5 运算符重载

运算符重载概念:对已有的运算符重新进行定义, 赋予其另一种功能, 以适应不同的数据类型

4.5.1 加号运算符重载

作用:实现两个自定义数据类型相加的运算

```
#include<iostream>
using namespace std;

class Person
{
public:
    int m_a;
    int m_b;

    //通过成员函数重载+
    /*Person operator+(Person& p)
    {
        Person temp;
        temp.m_a = this->m_a + p.m_a;
        temp.m_b = this->m_b + p.m_b;
        return temp;
    }*/

};

//通过全局函数重载+
Person operator+(Person& p1, Person& p2)
{
    Person temp;
    temp.m_a = p1.m_a + p2.m_a;
    temp.m_b = p1.m_b + p2.m_b;
    return temp;
}

Person operator+(Person& p, int a)
{
    Person temp;
    temp.m_a = p.m_a + a;
```

```
temp.m_b = p.m_b + a;
return temp;
}

void test01()
{
    Person p1;
    p1.m_a = 10;
    p1.m_b = 20;

    Person p2;
    p2.m_a = 10;
    p2.m_b = 20;

    //成员函数原理
    //Person p3 = p1.operator+(p2);

    //全局函数原理
    //Person p3 = operator+(p1, p2);
    //Person p3 = p1 + p2;

    //相当于Person p3 = operator+(p1,10);
    Person p3 = p1 + 10;

    cout << p3.m_a << endl;
    cout << p3.m_b << endl;
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

- 总结一：对于内置的数据类型的表达式的运算符是不可能改变的
- 总结二：不要滥用运算符重载

4.5.2 左移运算符重载

作用：可以输出自定义数据类型

```
#include<iostream>
using namespace std;

class Person
{
    friend ostream& operator<<(ostream& cout, Person& p);

public:
```

```

Person(int a, int b) :m_a(a), m_b(b){}
private:
    int m_a;
    int m_b;

};

//不能写成员函数原因, p.operator<<(cout)相当于p << cout 不是我们想要的

//想要实现链式法则输出, 就需要返回一个值cout
//cout<<p1 相当于 operator<<(cout,p)
ostream& operator<<(ostream &cout,Person &p)
{
    cout << "p.m_a = " << p.m_a << "  p.m_b = " << p.m_b << endl;
    return cout;
}

void test01()
{
    Person p1(10, 20);
    Person p2(20, 30);
    //链式法则
    cout << p1 << p2;
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

4.5.3 递增运算符重载

```

#include<iostream>
using namespace std;

class MyInteger
{
public:
    MyInteger()
    {
        m_num = 0;
    }
public:
    //前置递增运算符重载
    MyInteger& operator++()
    {
        ++m_num;
        return *this;
    }
}
```

```
}

//后置递增运算符重载
//void operator++(int) int代表占位参数，可以用于区分前置和后置递增
MyInteger operator++(int)
{
    //先记录当时的结果
    MyInteger temp = *this;
    //后递增
    m_num++;
    //最后将记录结果返回
    return temp;
}

int m_num;
};

//MyInteger myinteger这里不用引用的原因：后置递增运算符时返回值为局部变量temp，重载函数
//结束就会释放
// 非常量引用的初始值必须为左值（左值是指表达式结束后依然存在的持久对象）
ostream& operator<<(ostream& cout, MyInteger myinteger)
{
    cout << "m_num = " << myinteger.m_num << endl;
    return cout;
}

void test()
{
    MyInteger myinteger;
    cout << myinteger++ << endl;
    cout << "m_num = " << myinteger.m_num << endl;
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

4.5.4 赋值运算符重载

C++编译器至少给一个类添加4个函数

- 默认构造函数(无参，函数体为空)
- 默认析构函数(无参，函数体为空)
- 默认拷贝构造函数，对属性进行值拷贝
- 赋值运算符=，对属性进行值拷贝

如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题

```
#include<iostream>
using namespace std;

class Person
{
public:
    Person(int a)
    {
        m_age = new int(a);
    }

    Person(const Person &p)
    {
        m_age = new int(*p.m_age);
    }
    //考虑到链式法则实现 a = b = c,这里不用void operator=(Person& p)
    Person& operator=(Person& p)
    {
        //编译器是提供浅拷贝
        //m_age = p.m_age;

        //先判断是否有属性在堆区，有先释放干净，再深拷贝
        if (m_age != NULL)
        {
            delete m_age;
            m_age = NULL;
        }
        //深拷贝
        m_age = new int(*p.m_age);
        return *this;
    }

    ~Person()
    {
        if (m_age != NULL)
        {
            delete m_age;
            m_age = NULL;
        }
    }
    int *m_age;
};

void test()
{
    Person p1(18);
    Person p2(20);
    Person p3(30);
    p3 = p2 = p1;

    cout << "p1.m_a = " << *p1.m_age << endl;
    cout << "p2.m_a = " << *p2.m_age << endl;
    cout << "p3.m_a = " << *p3.m_age << endl;
}
```

```
}

int main()
{
    test();
    system("pause");
    return 0;
}
```

4.5.5 关系运算符重载

作用：重载关系运算符，可以让两个自定义类型对象进行对比操作

```
#include<iostream>
using namespace std;

class Person
{
public:
    Person(string name, int age)
    {
        m_name = name;
        m_age = age;
    }

    bool operator==(Person& p)
    {
        if (m_name == p.m_name && m_age == p.m_age)
        {
            return true;
        }
        return false;
    }

    string m_name;
    int m_age;
};

void test()
{
    Person p1("dc", 22);
    Person p2("dc", 20);

    if (p1 == p2)
    {
        cout << "p1和p2是同一个人" << endl;
    }
    else
    {
        cout << "p1和p2不是同一个人" << endl;
    }
}
```

```
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

4.5.6 函数调用运算符重载

- 函数调用运算符()也可以重载
- 由于重载后使用的方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

```
#include<iostream>
using namespace std;

class Myprint
{
public:
    //重载函数调用运算符
    void operator()(string test)
    {
        cout << test << endl;
    }
};

//仿函数非常灵活，没有固定的写法
//加法类

class Myadd
{
public:
    int operator()(int a, int b)
    {
        return a + b;
    }
};

void Myprint02(string test)
{
    cout << test << endl;
}

void test()
{
    Myprint myprint;

    myprint("hello world");//由于使用起来非常类似于函数调用，因此又称为：仿函数
```

```
Myprint02("hello world");//函数调用
}

void test02()
{
    Myadd myadd;

    int add = myadd(10, 20);
    cout << "add = " << add << endl;

    //匿名函数对象
    cout << Myadd()(100, 100) << endl;
}

int main()
{
    //test();
    test02();

    system("pause");
    return 0;
}
```

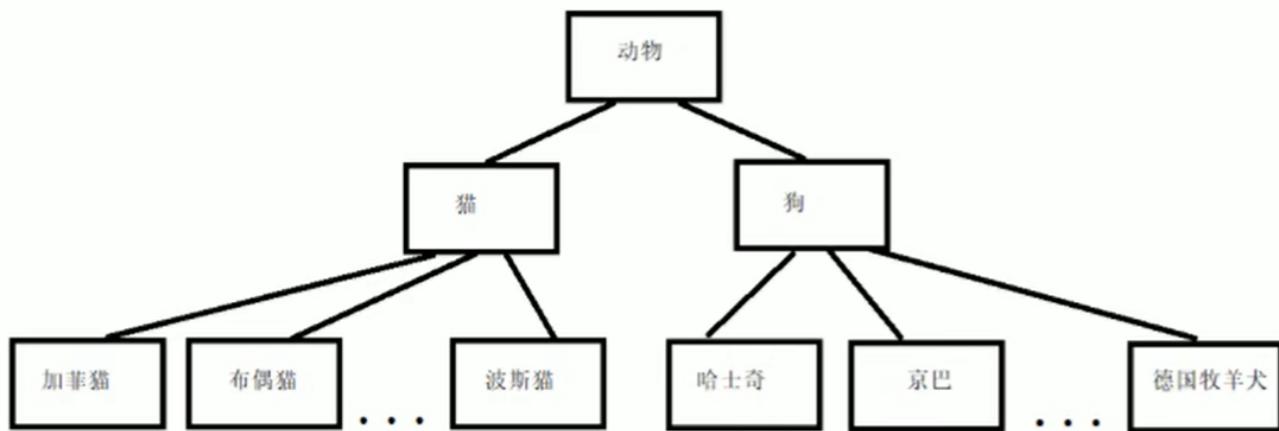
4.6 继承

继承是面向对象三大特性之一

语法: `class A:public B;`

- A类称为 子类或派生类
- B类称为 父类或基类

有些类与类之间存在特殊的关系, 例如下图中:



总结：

继承的好处：可以减少重复的代码

class A : public B;

A 类称为子类 或 派生类

B 类称为父类 或 基类

派生类中的成员，包含两大部分：

一类是从基类继承过来的，一类是自己增加的成员。

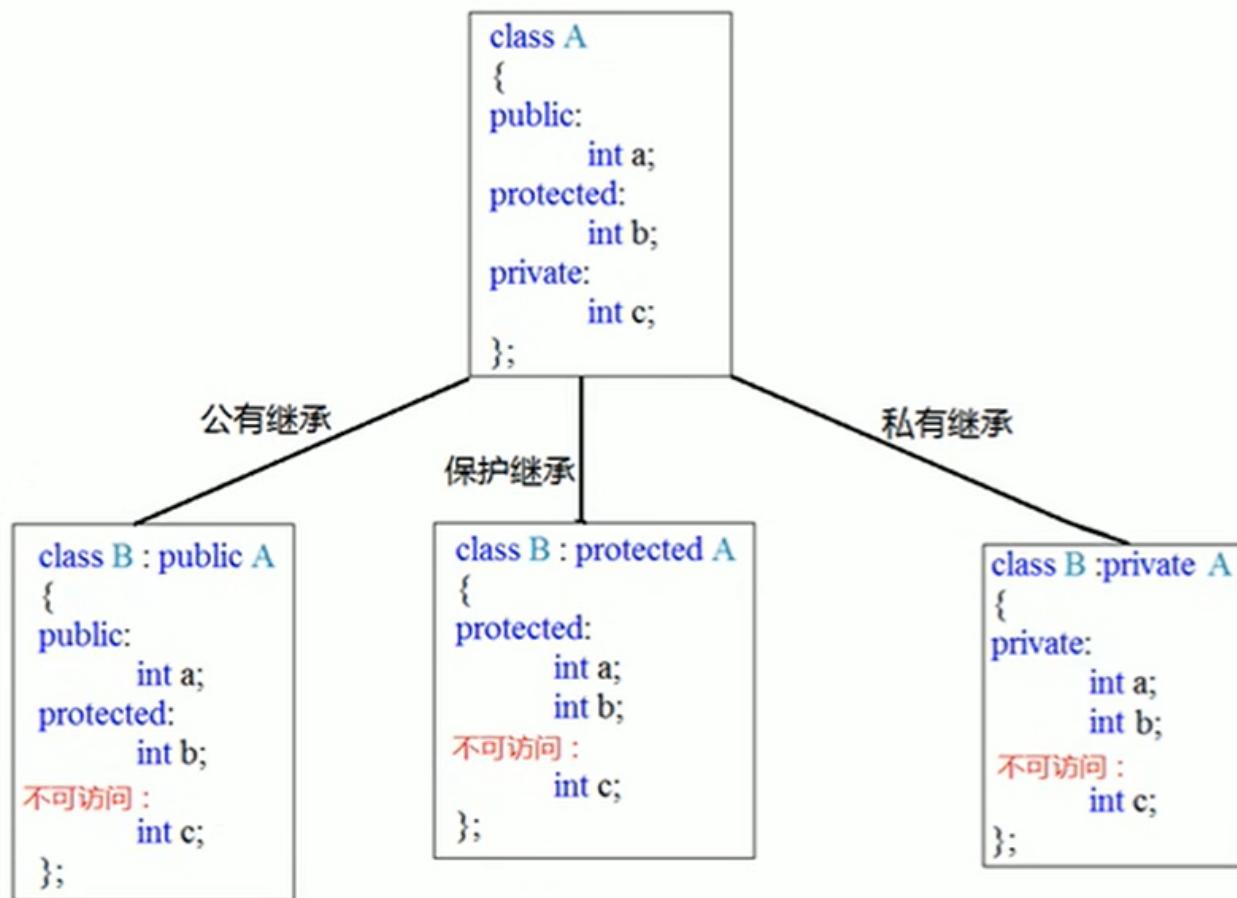
从基类继承过来的表现其共性，而新增的成员体现了其个性。

4.6.2 继承方式

继承的语法：class 子类：继承方式 父类

继承方式一共有三种：

- 公共继承
- 保护继承
- 私有继承



4.6.3 继承中的对象模型

```

#include<iostream>
using namespace std;

class Base
{
public:
    int m_a;
protected:
    int m_b;
private:
    int m_c; //私有成员只是被隐藏了，但是还是会继承下去
};

class son :public Base
{
public:
    int a;
};

void test()
{
    //16
    cout << "sizeof(son) = " << sizeof(son) << endl;
}

```

```

}

int main()
{
    test();
    system("pause");
    return 0;
}

```

利用开发人员命令提示工具查看对象模型

1. 跳转盘符
2. 跳转文件路径
3. dir
4. 查看命令 cl /d1 reportSingleClassLayout类名 文件名
5. 在我们的版本中查看命令这样写 (不会报错) : cl /EHsc /d1reportSingleClassLayout类名 文件名

D:\learningnote\C++\test02\test02>cl /d1 reportSingleClassLayoutson "10-03 继承中的对象模型 .cpp"
用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.41.34120 版
版权所有 (C) Microsoft Corporation。保留所有权利。

10-03 继承中的对象模型 .cpp

```

class son      size(16):
    +---+
0     | +--- (base class Base)
0     | | m_a
4     | | m_b
8     | | m_c
      | +---+
12    | a
      +---+

```

4.6.4 继承中的构造顺序

总结:继承中先调用父类构造函数, 再调用子类构造函数, 构顺序与构造相反

4.6.5 继承同名成员处理方式

问题:当子类与父类出现同名的成员, 如何通过子类对象, 访问到子类或父类中同名的数据呢?

- 访问子类同名成员直接访问即可
- 访问父类同名成员需要加作用域

```

#include<iostream>
using namespace std;

class Base
{
public:
    Base()
    {
        m_a = 100;
    }
}

```

```
void func()
{
    cout << "Base - func调用" << endl;
}

void func(int a)
{
    cout << "Base - func(int a)调用" << endl;
}

int m_a;
};

class Son :public Base
{
public:
    Son()
    {
        m_a = 200;
    }

    void func()
    {
        cout << "Son - func调用" << endl;
    }

    int m_a;
};

void test()
{
    Son s1;
    cout << "Son 下 m_a = " << s1.m_a << endl;
    //如果通过子类对象访问到父类中同名成员，需要加作用域
    cout << "Base 下 m_a = " << s1.Base::m_a << endl;
}

void test02()
{
    Son s2;
    s2.Base::func();

    //如果子类中出现和父类同名的成员函数，子类的同名成员会隐藏掉父类中所有同名成员函数
    s2.Base::func(100);
}

int main()
{
    test02();

    system("pause");
    return 0;
}
```

总结：

1. 子类对象可以直接访问到子类中同名成员
2. 子类对象加作用域可以访问到父类同名成员
3. 当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

4.6.6 继承同名静态成员处理

```
#include<iostream>
using namespace std;

class Base
{
public:

    static void func()
    {
        cout << "Base - func调用" << endl;
    }

    static void func(int a)
    {
        cout << "Base - func(int a)调用" << endl;
    }

    static int m_a;
};

int Base::m_a = 100;

class Son :public Base
{
public:

    static void func()
    {
        cout << "Son - func调用" << endl;
    }

    static int m_a;
};

int Son::m_a = 200;

//同名静态成员属性
void test()
{
    Son s1;
    cout << "通过对象访问" << endl;
    cout << "Son 下 static m_a = " << s1.m_a << endl;
    cout << "Base 下 static m_a = " << s1.Base::m_a << endl;
}
```

```
cout << "通过类名访问" << endl;
cout << "Son 下 static m_a =" << Son::m_a << endl;
cout << "Base 下 static m_a = " << Son::Base::m_a << endl;
}

//同名静态成员函数
void test02()
{
    Son s2;
    cout << "通过对象访问" << endl;
    s2.func();
    s2.Base::func();
    s2.Base::func(100);

    cout << "通过类名访问" << endl;
    Son::func();
    Son::Base::func();
    Son::Base::func(100);
}

int main()
{
    test02();

    system("pause");
    return 0;
}
```

总结:同名静态成员处理方式和非静态处理方式一样，只不过有两种访问的方式(通过对象和通过类名)

4.6.7 多继承语法

C++允许一个类继承多个类

语法: `class 子类: 继承方式 父类1, 继承方式, 父类2...`

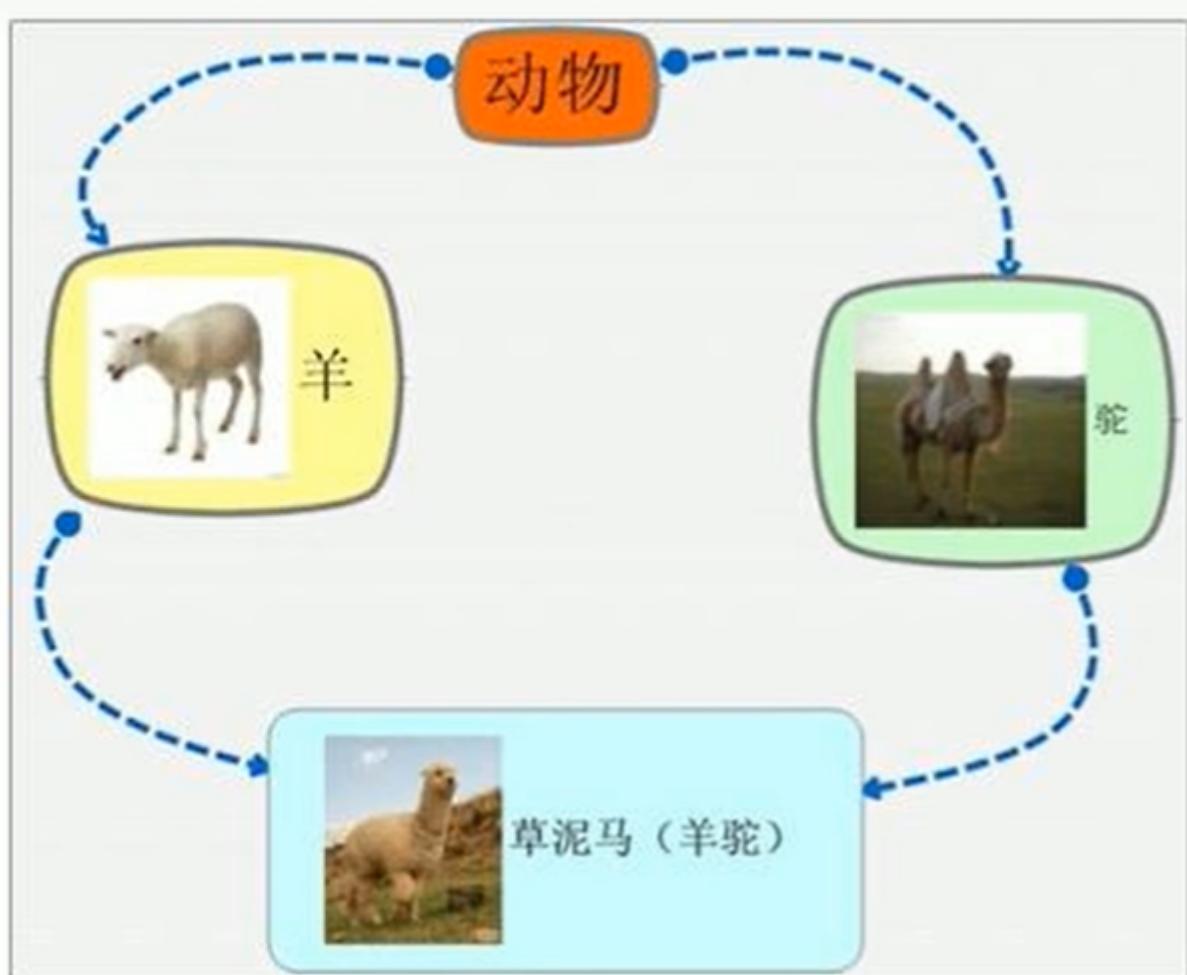
多继承可能会引发父类中有同名成员出现，需要加作用域区分

C++实际开发中不建议用多继承

4.6.8 菱形继承

菱形继承概念：

- 两个派生类继承同一个基类
- 又有某个类同时继承者两个派生类
- 这种继承被称为菱形继承，或者钻石继承



菱形继承

承问题：

1. 羊继承了动物的数据，驼同样继承了动物的数据，当草泥马使用数据时，就会产生二义性.
2. 草泥马继承自动物的数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以.

```

#include<iostream>
using namespace std;

class Animals
{
public:
    int m_Age;
};

//利用虚继承解决菱形继承的问题 virtual
//继承之前加上关键字虚拟变为虚继承
//动物类称为虚基类
class Sheep:virtual public Animals{};

class Tuo :virtual public Animals{};

class Sheeptuo :public Sheep, public Tuo{};

void test()
  
```

```

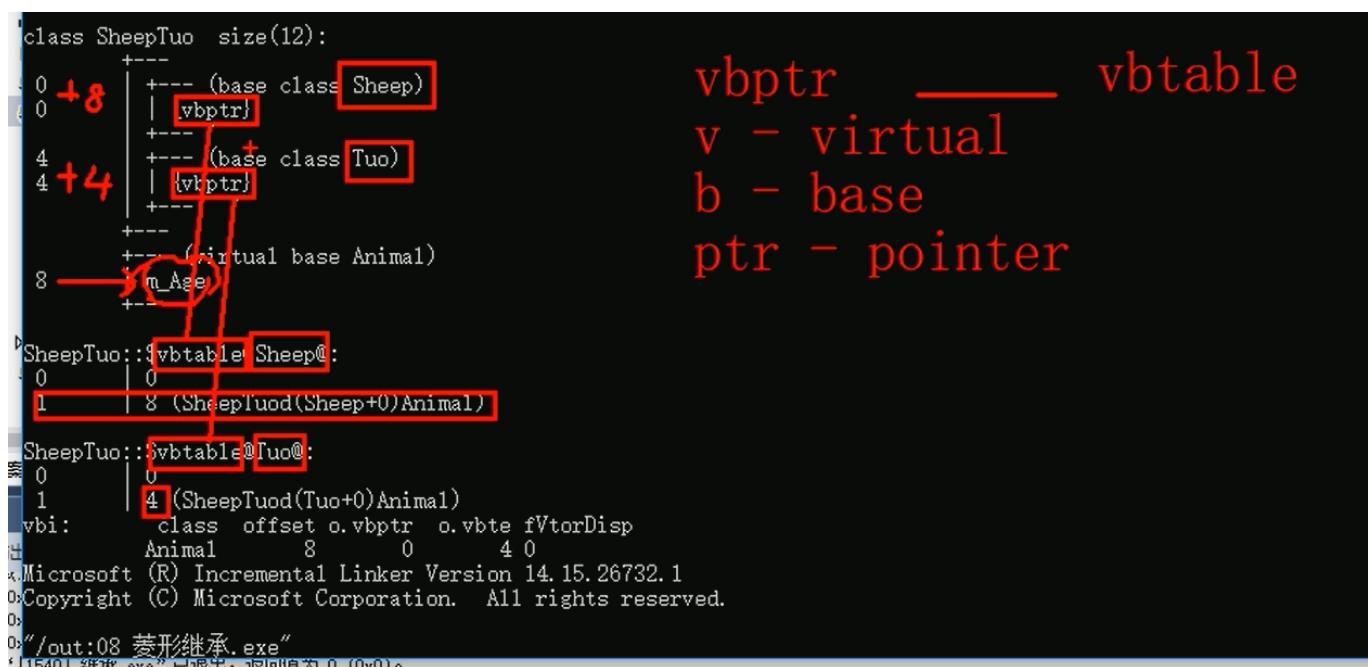
{
    Sheeptuo s1;

    s1.Sheep::m_Age = 18;
    s1.Tuo::m_Age = 28;
    //当菱形继承，两个父类拥有相同数据，需要加以作用域区分
    cout << "s1.Sheep::m_Age = " << s1.Sheep::m_Age << endl;
    cout << "s1.Tuo::m_Age = " << s1.Tuo::m_Age << endl;
    cout << "s1.m_Age = " << s1.m_Age << endl;
}

int main()
{
    test();

    system("pause");
    return 0;
}

```



总结：

- 菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题

4.7 多态

多态是C++面向对象三大特性之一

多态分为两类

- 静态多态：函数重载和运算符重载属于静态多态，复用函数名
- 动态多态：派生类和虚函数实现运行时多态

静态多态和动态多态区别：

- 静态多态的函数地址早绑定-编译阶段确定函数地址
- 动态多态的函数地址晚绑定-运行阶段确定函数地址

4.7.1 多态的基本概念

```
#include<iostream>
using namespace std;

class Animals
{
public:
    virtual void speak()
    {
        cout << "动物在说话" << endl;
    }
};

class Cat:public Animals
{
public:
    //重写 函数返回值类型    函数名 参数列表 完全相同
    void speak()
    {
        cout << "猫在说话" << endl;
    }
};

class Dog :public Animals
{
public:
    void speak()
    {
        cout << "狗在说话" << endl;
    }
};

//执行说话的函数
//地址早绑定在编译阶段确定函数地址
//如果想执行让猫说话，那么这个函数地址就不能提前绑定，需要在运行阶段进行绑定，地址晚绑定

//动态多态满足条件
//1、有继承关系
//2、子类重写父类的虚函数

//动态多态使用
//父类的指针或者引用指向子类对象 Animals& animals

void doSpeak(Animals& animals)
{
    animals.speak();
```

```
}

void test()
{
    Cat cat;
    doSpeak(cat);

    Dog dog;
    doSpeak(dog);
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

总结：

多态满足条件：

- 有继承关系
- 子类重写父类中的虚函数

多态使用条件：

- 父类指针或引用指向子类对象

重写：函数返回值类型函数名参数列表完全一致称为重写

4.7.2 多态案例--计算机类

案例描述：

分别利用普通写法和多态技术，设计实现两个操作数进行运算的计算器类

多态的优点：

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展以及维护

4.7.3 纯虚函数和抽象类

在多态中，通常父类中虚函数的实现是毫无意义的，主要都是调用子类重写的内容，因此可以将虚函数改为纯虚函数

纯虚函数语法：`virtual 返回值类型 函数名 (参数列表)=0;`

当类中有了纯虚函数，这个类也称为**抽象类**

抽象类特点:

- 无法实例化对象
- 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

4.7.4 虚析构和纯虚析构

多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码

解决方式：将父类中的析构函数改为虚析构或者纯虚析构

虚析构和纯虚析构共性：

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

虚析构和纯虚析构区别：

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

虚析构语法：`virtual ~类名() {}`

纯虚析构语法：`virtual ~类名() = 0 ; 类名::~类名() {}`

```
#include<iostream>
using namespace std;

class Animals
{
public:
    Animals()
    {
        cout << "animals的构造函数调用" << endl;
    }
    //虚析构
    /*virtual ~Animals()
    {
        cout << "animals的析构函数调用" << endl;
    }*/
    //纯虚析构
    virtual ~Animals() = 0;

    virtual void Speak()
    {
        cout << "动物在说话" << endl;
    }
};

//纯虚析构也需要一个实现
Animals::~Animals()
{
    cout << "animals的析构函数调用" << endl;
}
```

```
class Cat:public Animals
{
public:
    Cat(string name)
    {
        cout << "cat的构造函数调用" << endl;
        m_name = new string(name);
    }

    ~Cat()
    {
        if (m_name != NULL)
        {
            cout << "cat的析构函数调用" << endl;
            delete m_name;
            m_name = NULL;
        }
    }

    virtual void Speak()
    {
        cout << *m_name << "猫在说话" << endl;
    }

    string *m_name;
};

void test01()
{
    Animals* a1 = new Cat("Tom");
    a1->Speak();
    delete a1;
}

int main()
{
    test01();

    system("pause");
    return 0;
}
```

5 文件操作

程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放

通过文件可以将数据持久化

C++中对文件操作需要包含头文件 <fstream>

文件类型分为两种：

1. **文本文件** - 文件以文本的ASCII码形式存储在计算机中
2. **二进制文件** - 文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂它们

操作文件的三大类：

1. ofstream：写操作
2. ifstream：读操作
3. fstream：读写操作

5.1 文本文件

5.1.1 写文件

写文件步骤如下：

1. 包含头文件：#include<fstream>
2. 创建流对象：ofstream ofs;
3. 打开文件：ofs.open("文件路径", 打开方式);
4. 写数据：ofs<<"写入的数据";
5. 关闭文件：ofs.close();

文件打开方式：

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置：文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在，先删除，再创建
ios::binary	二进制方式

注意：文件打开方式可以配合使用，利用 | 操作符

例如：用二进制方式写文件`ios::binary | ios::out`

5.1.2 读文件

读文件与写文件步骤相似，但是读取方式相对于比较多

读文件步骤如下：

1. 包含头文件:`#include<fstream>`
2. 创建流对象:`ifstream ifs;`
3. 打开文件并判断文件是否打开成功:`ifs.Open("文件路径", 打开方式);`
4. 读数据:四种方式读取
5. 关闭文件:`ifs.close()`

```
#include<iostream>
using namespace std;
#include<fstream>
#include<string>

void test()
{
    //2.创建流对象
    ifstream ifs;

    //3.打开文件并判断是否打开成功
    ifs.open("test.txt", ios::in);

    if (!ifs.is_open())
    {
        cout << "文件打开失败，文件不存在" << endl;
        return;
    }
    //4.读入数据
    //第一种
    //char buf[1024] = { 0 };
    //while (ifs >> buf)
    //{
    //    cout << buf << endl;
    //}

    //第二种
    //char buf[1024] = { 0 };
    //while (ifs.getline(buf, sizeof(buf)))
    //{
    //    cout << buf << endl;
    //}

    //第三种
    string buf;
    while (getline(ifs, buf))
    {
        cout << buf << endl;
    }
}
```

```
}

//第四种方法
char c;
while ((c = ifs.get() != EOF))//EOF: end of file文件结尾符
{
    cout << c;
}

//5.关闭文档
ifs.close();
}

int main()
{
    test();
    system("pause");
    return 0;
}
```

总结：

- 读文件可以利用ifstream或者fstream类
- 利用is_open函数可以判断文件是否打开成功
- 关闭关闭文件

C++提高编程

1. 模版

本阶段主要针对C++泛型编程和stl技术做详细讲解，探讨C++更深层的使用

1.1 模版的概念

模版就是建立通用的模具，大大提高复用性

1.2 函数模版

C++另一种编程思想称为泛型编程

- 主要利用的技术就是模板
- C++提供两种模板机制：**函数模板**和**类模板**

1.2.1 函数模版语法

函数模板作用：

建立一个通用函数，其函数返回值类型和形参类型可以不具体制定，用一个虚拟的类型来代表

语法：

```
template<typename T>
函数定义或声明
```

解释：

template ... 声明创建模板

typename ... 表面其后面的符号是一种数据类型，可以用class代替

T ... 通用的数据类型，名称可以替换，通常为大写字母

总结：

- 函数模板利用关键字模板
- 使用函数模板有两种方式：自动类型推导、显示指定类型
- 模板的目的是为了提高复用性，将类型参数代

1.2.2 函数模版注意事项

注意事项：

- 自动类型推导，必须推导出一致的数据类型T，才可以使用
- 模板必须要确定出T的数据类型，才可以使用

1.2.3 普通函数和函数模版的区别

普通函数与函数模板区别：

- 普通函数调用时可以发生自动类型转换(隐式类型转换)
- 函数模板调用时，如果利用自动类型推导，不会发生隐式类型转换
- 如果利用显示指定类型的方式，可以发生隐式类型转换

1.2.4 普通函数和函数模版的调用规则

调用规则如下

1. 如果函数模板和普通函数都可以实现，优先调用普通函数
2. 可以通过空模板参数列表来强制调用函数模板
3. 函数模板也可以发生重载
4. 如果函数模板可以产生更好的匹配，优先调用函数模板

1.2.4 函数模版的局限性

模板的通用性并不是万能的

例如：

```
1 template<class T>
2 void f(T a, T b)
3 {
4     a = b;]
5 }
```

在上述代码中提供的赋值操作，如果传入的a和b是一个数组，就无法实现了

在上述代码中，如果T的数据类型传入的是像Person这样的自定义数据类型，也无法正常运行

因此C++为了解决这种问题，提供模板的重载，可以为这些特定的类型提供具体化的模板

//利用具体化Person的版本实现代码，具体化优先调用
template<> bool myCompare(Person &p1, Person &p2)
{
}
}

总结：

- 利用具体化的模板，可以解决自定义类型的通用化
- 学习模板并不是为了写模板，而是在STL能够运用系统提供的模板

1.3 类模版

1.3.1 类模板语法

建立一个通用类，类中的成员数据类型可以不具体制定，用一个虚拟的类型来代表

语法：

```
template<typename T>
类
```

解释：

template --- 声明创建模板

typename --- 表面其后面的符号是一种数据类型，可以用class代替

T --- 通用的数据类型，名称可以替换，通常为大写字母

类模版和函数模版语法相似，在声明模版template后面加类，此类称为模版

1.3.2 类模板与函数模版区别

类模板与函数模板区别主要有两点：

1. 类模板没有自动类型推导的使用方式
2. 类模板在模板参数列表中可以有默认参数

1.3.3 类模版中成员函数创建时机

类模板中成员函数和普通类中成员函数创建时机是有区别的

- 普通类中的成员函数一开始就可以创建
- 类模板中的成员函数在调用时才创建

总结：类模版中的成员函数并不是一开始就创建的，在调用时才去创建。

1.3.4 类模版对象做函数参数

一共有三种传入方式：

1. 指定传入的类型 --直接显示对象的数据类型
2. 参数模板化 --将对象中的参数变为模板进行传递
3. 整个类模板化 --将这个对象类型模板化进行传递

```
#include<iostream>
using namespace std;
#include<string>

template<class T1, class T2>
class Person
{
public:
    Person(T1 name, T2 age)
    {
        this->m_name = name;
        this->m_age = age;
    }

    void showperson()
```

```
{  
    cout << "姓名: " << this->m_name << "\t年龄: " << this->m_age << endl;  
}  
  
T1 m_name;  
T2 m_age;  
};  
  
//1.指定传入类型  
void printperson1(Person<string, int>&p)  
{  
    p.showperson();  
}  
  
void test01()  
{  
    Person<string, int>p1("孙悟空", 100);  
    printperson1(p1);  
}  
  
//2.参数模板化  
template<class T1, class T2>  
void printperson2(Person<T1, T2>& p)  
{  
    p.showperson();  
    cout << "T1的类型为" << typeid(T1).name() << endl;  
    cout << "T2的类型为" << typeid(T2).name() << endl;  
}  
  
void test02()  
{  
    Person<string, int>p2("猪八戒", 90);  
    printperson2(p2);  
}  
  
//3.整个类模版化  
template<class T>  
void printperson3(T& p)  
{  
    p.showperson();  
    cout << "T的类型为" << typeid(T).name() << endl;  
}  
  
void test03()  
{  
    Person<string, int>p3("唐僧", 20);  
    printperson3(p3);  
}  
  
int main()  
{  
    test03();  
  
    system("pause");  
}
```

```
    return 0;
}
```

总结：

- 通过类模板创建的对象，可以有三种方式向函数中进行传参
- 使用比较广泛是第一种：指定传入的类型

1.3.5 类模版与继承

当类模板碰到继承时，需要注意一下几点

- 当子类继承的父类是一个类模板时，子类在声明的时候，要指定出类中T的类型
- 如果不指定，编译器无法给子类分配内存
- 如果想灵活指定出父类中T的类型，子类也需变为类模板

总结：如果父类是类模板，子类需要指定出父类中T的数据类型

1.3.6 类模版成员函数类外实现

```
#include<iostream>
using namespace std;
#include<string>

template<class T1, class T2>
class People
{
public:
    People(T1 name, T2 age);
    //{
    //    this->m_name = name;
    //    this->m_age = age;
    //}

    void showperson();
    //{
    //    cout << "姓名：" << this->m_name << "\t年龄：" << this->m_age << endl;
    //}

    T1 m_name;
    T2 m_age;
};

//构造函数类外实现
template<class T1, class T2>
People<T1,T2>::People(T1 name, T2 age)
{
    this->m_name = name;
    this->m_age = age;
}
```

```
//成员函数类外实现
template<class T1, class T2>
void People<T1,T2>::showperson()
{
    cout << "姓名: " << this->m_name << "\t年龄: " << this->m_age << endl;
}

void test()
{
    People<string, int>p1("ss", 10);
    p1.showperson();
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

1.3.7 类模版分文件编写

学习目标：

- 掌握类模板成员函数分文件编写产生的问题以及解决方式

问题：

- 类模板中成员函数创建时机是在调用阶段，导致分文件编写时链接不到

解决：

- 解决方式1：直接包含.cpp源文件
- 解决方式2：将声明和实现写到同一个文件中，并更改后缀名为.hpp，.hpp是约定的名称，并不是强制

1.3.8 类模版与友元

学习目标：

- 掌握类模板配合友元函数的类内和类外实现

全局函数类内实现 - 直接在类内声明友元即可

全局函数类外实现 - 需要提前让编译器知道全局函数的存在

总结:建议全局函数做类内实现，用法简单，而且编译器可以直接识别

2. STL初识

2.1 STL的诞生

- 长久以来，软件界一直希望建立一种可重复利用的东西
- C++的面向对象和泛型编程思想，目的就是复用性的提升
- 大多情况下，数据结构和算法都未能有一套标准，导致被迫从事大量重复工作
- 为了建立数据结构和算法的一套标准，诞生了STL

2.2 STL基本概念

- STL(Standard Template Library)标准模板库
- STL从广义上分为:容器(容器)算法(算法)迭代器(迭代器)
- 容器和算法之间通过迭代器进行无缝连接.
- STL几乎所有的代码都采用了模板类或者模板函数

2.3 STL六大组件

STL大体分为六大组件，分别是:容器、算法、迭代器、仿函数、适配器(接器)、空间配置器

1. 容器:各种数据结构，如vector、List、deque、set、map等，用来存放数据
2. 算法:各种常用的算法，如Sort、Find、Copy、For_Each等
3. 迭代器:扮演了容器与算法之间的胶合剂
4. 仿函数:行为类似函数，可作为算法的某种策略
5. 适配器:一种用来修饰容器或者仿函数或迭代器接口的东西
6. 空间配置器:负责空间的配置与管理

2.4 STL中容器、算法、迭代器

容器:置物之所也

STL容器就是将运用最广泛的一些数据结构实现出来

常用的数据结构:数组，链表，树，栈，队列，集合，映射表等

这些容器分为序列式容器和关联式容器两种：

- 序列式容器：强调值的排序，序列式容器中的每个元素均有固定的位置，
- 关联式容器：二叉树结构，各元素之间没有严格的物理上的顺序关系

算法：问题之解法也

有限的步骤，解决逻辑或数学上的问题，这一门学科我们叫做算法(Algorithms)

算法分为：质变算法和非质变算法

- 质变算法：是指运算过程中会更改区间内的元素的内容。例如拷贝，替换删除等等
- 非质变算法：是指运算过程中不会更改区间内的元素内容，例如查找、计数、遍历、寻找极值等等

迭代器：容器和算法之间粘合剂

提供一种方法，使之能够依序寻访某个容器所含的各个元素，而又无需暴露该容器的内部表示方式

- 每个容器都有自己专属的迭代器
- 迭代器使用非常类似于指针，初学阶段我们可以先理解迭代器为指针

迭代器种类：

种类	功能	支持运算
输入迭代器	对数据的只读访问	只读，支持++、==、!=
输出迭代器	对数据的只写访问	只写，支持++
前向迭代器	读写操作，并能向前推进迭代器	读写，支持++、==、!=
双向迭代器	读写操作，并能向前和向后操作	读写，支持++、--、
随机访问迭代器	读写操作，可以以跳跃的方式访问任意数据，功能最强的迭代器	读写，支持++、--、[n]、-n、<、<=、>、>=

常用的容器中迭代器种类为双向迭代器，和随机访问迭代器

2.5 容器、算法、迭代器初识

STL中最常用的容器为Vector，可以理解为数组，下面我们将学习如何向这个容器中插入数据、并遍历这个容器

2.5.1 vector存放内置数据类型

容器：`vector`

算法：`for_each`

迭代器：`vector<int>::iterator`

```
using namespace std;
#include<vector>
#include<algorithm>

void myprint(int val)
{
    cout << val << endl;
}

void test01()
{
    //创建一个vector容器数组
    vector<int> v;

    //向容器中插入数据
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    ////通过迭代器访问容器中的数据
    //vector<int>::iterator itBegin = v.begin(); //起始迭代器 指向容器中第一个元素
    //vector<int>::iterator itEnd = v.end(); //起始迭代器 指向容器中最后一个元素的
    下一个位置

    ////第一种遍历方式
    //while (itBegin != itEnd)
    //{
    //    cout << *itBegin << endl;
    //    itBegin++;
    //}

    ////第二种遍历方式
    //for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    //{
    //    cout << *it << endl;
    //}

    //第三种遍历方式
    for_each(v.begin(), v.end(), myprint);
}

int main()
{
    test01();

    system("pause");
    return 0;
}
```

2.5.2 vector存放自定义数据类型

```
#include<iostream>
using namespace std;
#include<vector>

class Person
{
public:
    Person(string name, int age)
    {
        this->m_name = name;
        this->m_age = age;
    }

    string m_name;
    int m_age;
};

void test01()
{
    vector<Person>v;

    Person p1("aaa", 10);
    Person p2("bbb", 12);
    Person p3("ccc", 13);
    Person p4("ddd", 14);
    Person p5("eee", 15);

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);
    v.push_back(p5);

    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << "姓名: " << it->m_name << "年龄: " << it->m_age << endl;
    }
}

//存放自定义数据类型的指针
void test02()
{
    vector<Person*>v;

    Person p6("aaa", 10);
    Person p7("bbb", 12);
    Person p8("ccc", 13);
    Person p9("ddd", 14);
```

```
Person p10("eee", 15);

v.push_back(&p6);
v.push_back(&p7);
v.push_back(&p8);
v.push_back(&p9);
v.push_back(&p10);

for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++)
{
    cout << "姓名: " << (*it)->m_name << "年龄: " << (*it)->m_age << endl;
}
}

int main()
{
    test02();

    system("pause");
    return 0;
}
```

2.5.3 vector容器嵌套容器

```
#include<iostream>
using namespace std;
#include<vector>

void test()
{
    vector<vector<int>>v;

    vector<int>v1;
    vector<int>v2;
    vector<int>v3;
    vector<int>v4;
    vector<int>v5;

    for (int i = 0; i < 5; i++)
    {
        v1.push_back(i + 1);
        v2.push_back(i + 2);
        v3.push_back(i + 3);
        v4.push_back(i + 4);
        v5.push_back(i + 5);
    }

    v.push_back(v1);
    v.push_back(v2);
    v.push_back(v3);
    v.push_back(v4);
```

```
v.push_back(v5);

for (vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++)
{
    // (*it) 代表vector<int>类型, 再遍历
    for (vector<int>::iterator vit = (*it).begin(); vit != (*it).end(); vit++)
    {
        cout << (*vit) << " ";
    }
    cout << endl;
}
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

3 STL常用容器

3.1 string容器

3.1.1 string基本概念

本质:

- string是C++风格的字符串, 而string本质上是一个类
-

string和char*区别:

- char*是一个指针
- string是一个类, 类内部封装了char*, 理这个字符串, 是一个char*型的容器。

特点:

- string类内部封装了很多成员方法

例如: 查找find, 拷贝copy, 删除delete替换replace, 插入insert

string管理char*所分配的内存, 不用担心复制越界和取值越界等, 由类内部进行负责

3.1.2 string构造函数

构造函数原型:

- `string();` //创建一个空的字符串 例如: string str;
- `string(const char* s);` //使用字符串s初始化
- `string(const string& str);` //使用一个string对象初始化另一个string对象
- `string(int n, char c);` //使用n个字符c初始化

```
#include<iostream>
using namespace std;
#include<string>

void test()
{
    string s1;

    const char* str = "hello world";
    string s2(str);
    cout << "s2 = " << s2 << endl;

    string s3(s2);
    cout << "s3 = " << s3 << endl;

    string s4(10, 'a');
    cout << "s4 = " << s4 << endl;
}

int main()
{
    test();

    system("pause");
    return 0;
}
```

总结:string的多种构造方式没有可比性，灵活使用即可

3.1.3 string赋值操作

功能描述：

- 给string字符串进行赋值

赋值的函数原型：

- `string& operator=(const char* s);` //char*类型字符串 赋值给当前的字符串
- `string& operator=(const string &s);` //把字符串s赋给当前的字符串
- `string& operator=(char c);` //字符赋值给当前的字符串
- `string& assign(const char *s);` //把字符串s赋给当前的字符串
- `string& assign(const char *s, int n);` //把字符串s的前n个字符赋给当前的字符串
- `string& assign(const string &s);` //把字符串s赋给当前字符串
- `string& assign(int n, char c);` //用n个字符c赋给当前字符串

```
#include<iostream>
using namespace std;
#include<string>

//string& operator=(const char* s);
//string& operator=(const string& s);
//string& operator=(char c);
//string& assign(const char* s);
//string& assign(const char* s, int n);
//string& assign(const string& s);
//string& assign(int n, char c);

void test()
{
    string s1;
    s1 = "hello world";
    cout << "s1 = " << s1 << endl;

    string s2;
    s2 = s1;
    cout << "s2 = " << s2 << endl;

    string s3;
    s3 = 'a';
    cout << "s3 = " << s3 << endl;

    string s4;
    s4.assign("hello world");
    cout << "s4 = " << s4 << endl;

    string s5;
```

```
s5.assign("hello world", 5);
cout << "s5 = " << s5 << endl;

string s6;
s6.assign(s5);
cout << "s6 = " << s6 << endl;

string s7;
s7.assign(10, 'w');
cout << "s7 = " << s7 << endl;
}

int main()
{
    test();
    system("pause");
    return 0;
}
```

总结:

string的赋值方式很多，`operator=`这种方式是比较实用的

3.1.4 string字符串拼接

功能描述:

- 实现在字符串末尾拼接字符串

函数原型:

- `string& operator+=(const char* str);` //重载+=操作符
- `string& operator+=(const char c);` //重载+=操作符
- `string& operator+=(const string& str);` //重载+=操作符
- `string& append(const char *s);` //把字符串s连接到当前字符串结尾
- `string& append(const char *s, int n);` //把字符串s的前n个字符连接到当前字符串结尾
- `string& append(const string &s);` //同operator+=(const string& str)
- `string& append(const string &s, int pos, int n);` //字符串s中从pos开始的n个字符连接到字符串结尾

3.1.5 string查找和替换

功能描述:

- 查找: 查找指定字符串是否存在
- 替换: 在指定的位置替换字符串

函数原型:

- | | |
|---|-------------------------|
| • int find(const string& str, int pos = 0) const; | //查找str第一次出现位置,从pos开始查找 |
| • int find(const char* s, int pos = 0) const; | //查找s第一次出现位置,从pos开始查找 |
| • int find(const char* s, int pos, int n) const; | //从pos位置查找s的前n个字符第一次位置 |
| • int find(const char c, int pos = 0) const; | //查找字符c第一次出现位置 |
| • int rfind(const string& str, int pos = npos) const; | //查找str最后一次位置,从pos开始查找 |
| • int rfind(const char* s, int pos = npos) const; | //查找s最后一次出现位置,从pos开始查找 |
| • int rfind(const char* s, int pos, int n) const; | //从pos查找s的前n个字符最后一次位置 |
| • int rfind(const char c, int pos = 0) const; | //查找字符c最后一次出现位置 |
| • string& replace(int pos, int n, const string& str); | //替换从pos开始n个字符为字符串str |
| • string& replace(int pos, int n, const char* s); | //替换从pos开始的n个字符为字符串s |

```
#include<iostream>
using namespace std;
#include<string>

//1.查找
void test()
{
    string s1;
    s1 = "abcdefgde";

    int pos = s1.find("de");
    if (pos == -1)
    {
        cout << "查找内容不存在" << endl;
    }
    else
    {
        cout << "pos = " << pos << endl;
    }

    //rfind 从右向左查
    pos = s1.rfind("de");
    if (pos == -1)
    {
        cout << "查找内容不存在" << endl;
    }
    else
    {
        cout << "pos = " << pos << endl;
    }
}
```

```
}

//2.替换
void test02()
{
    string s2;
    s2 = "abcdefgde";

    //从1号位置开始 3个字符被替换为1111
    s2.replace(1, 3, "1111");
    cout << s2 << endl;
}

int main()
{
    test02();

    system("pause");
    return 0;
}
```

总结:

- find查找是从左往右， rfind从右往左
- find找到字符串后返回查找的第一个字符位置，找不到返回-1
- replace在替换时，要指定从哪个位置起，多少个字符，替换成什么样的字符串

3.1.6 string字符串比较

功能描述：

- 字符串之间的比较

比较方式：

- 字符串比较是按字符的ASCII码进行对比

= 返回 0

> 返回 1

< 返回 -1

I

函数原型：

- `int compare(const string &s) const;` //与字符串s比较
- `int compare(const char *s) const;` //与字符串s比较

3.1.7 string字符存取

string中单个字符存取方式有两种

- `char& operator[](int n);` //通过[]方式取字符
- `char& at(int n);` //通过at方法获取字符

3.1.8 string插入和删除

功能描述：

- 对string字符串进行插入和删除字符操作

函数原型：

- `string& insert(int pos, const char* s);` //插入字符串
- `string& insert(int pos, const string& str);` //插入字符串
- `string& insert(int pos, int n, char c);` //在指定位置插入n个字符c
- `string& erase(int pos, int n = npos);` //删除从Pos开始的n个字符

总结：插入和删除的起始下标都是从0开始

3.1.8 string子串

功能描述：

- 从字符串中获取想要的子串

函数原型：

- `string substr(int pos = 0, int n = npos) const;` //返回由pos开始的n个字符组成的字符串

3.2 vector容器

3.2.1 vector基本概念

功能：

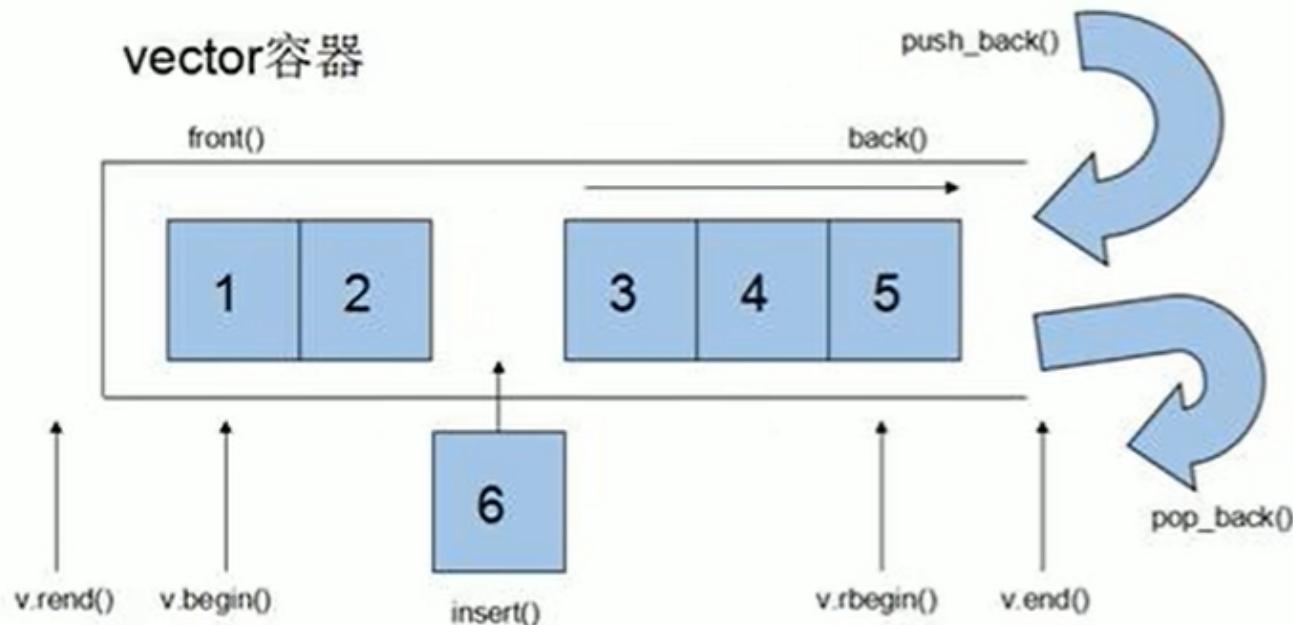
- vector数据结构和数组非常相似，也称为单端数组

vector与普通数组区别：

- 不同之处在于数组是静态空间，而vector可以动态扩展

动态扩展：

- 并不是在原空间之后续接新空间，而是找更大的内存空间，然后将原数据拷贝新空间，释放原空间



- vector容器的迭代器是支持随机访问的迭代器

3.2.2 vector构造函数

功能描述:

- 创建vector容器

函数原型:

- `vector<T> v;` //采用模板实现类实现， 默认构造函数
- `vector(v.begin(), v.end());` //将v[begin(), end())区间中的元素拷贝给本身。
- `vector(n, elem);` //构造函数将n个elem拷贝给本身。
- `vector(const vector &vec);` //拷贝构造函数。

3.2.3 vector赋值操作

功能描述：

- 给vector容器进行赋值

函数原型：

- `vector& operator=(const vector &vec);` //重载等号操作符
- `assign(beg, end);` //将[beg, end)区间中的数据拷贝赋值给本身。
- `assign(n, elem);` //将n个elem拷贝赋值给本身。

3.2.4 vector容量大小

功能描述：

- 对vector容器的容量和大小操作

函数原型：

- `empty();` //判断容器是否为空
- `capacity();` //容器的容量
- `size();` //返回容器中元素的个数
- `resize(int num);` //重新指定容器的长度为num，若容器变长，则以默认值填充新位置。
//如果容器变短，则末尾超出容器长度的元素被删除。
- `resize(int num, elem);` //重新指定容器的长度为num，若容器变长，则以elem值填充新位置。
//如果容器变短，则末尾超出容器长度的元素被删除

3.2.5 vector插入和删除

我也在，看元

功能描述：

- 对vector容器进行插入、删除操作

函数原型：

- `push_back(ele);` //尾部插入元素ele
- `pop_back();` //删除最后一个元素
- `insert(const_iterator pos, ele);` //迭代器指向位置pos插入元素ele
- `insert(const_iterator pos, int count, ele);` //迭代器指向位置pos插入count个元素ele
- `erase(const_iterator pos);` //删除迭代器指向的元素
- `erase(const_iterator start, const_iterator end);` //删除迭代器从start到end之间的元素
- `clear();` //删除容器中所有元素

总结：

- 尾插 --- `push_back`
- 尾删 --- `pop_back`
- 插入 --- `insert` (位置迭代器)
- 删除 --- `erase` (位置迭代器)
- 清空 --- `clear`

3.2.6 vector数据存取

功能描述：

- 对vector中的数据的存取操作

函数原型：

- `at(int idx);` //返回索引idx所指的数据
- `operator[]();` //返回索引idx所指的数据
- `front();` //返回容器中第一个数据元素
- `back();` //返回容器中最后一个数据元素

3.2.7 vector互换容器

功能描述：

- 实现两个容器内元素进行互换

函数原型：

- `swap(vec);` // 将vec与本身的元素互换

```
//vector互换容器示例，收缩内存
void test05()
{
    vector<int>v1;

    for (int i = 0; i < 10000; i++)
    {
        v1.push_back(i);
    }

    cout << "v1的容量为：" << v1.capacity() << endl;
    cout << "v1的大小为：" << v1.size() << endl;
```

```
v1.resize(3);
cout << "v1的容量为: " << v1.capacity() << endl;
cout << "v1的大小为: " << v1.size() << endl;

vector<int>(v1).swap(v1);
cout << "v1的容量为: " << v1.capacity() << endl;
cout << "v1的大小为: " << v1.size() << endl;
}
```

3.2.8 vector预留空间

功能描述:

- 减少vector在动态扩展容量时的扩展次数

函数原型:

- `reserve(int len);` //容器预留len个元素长度，预留位置不初始化，元素不可访问。

3.3 deque容器

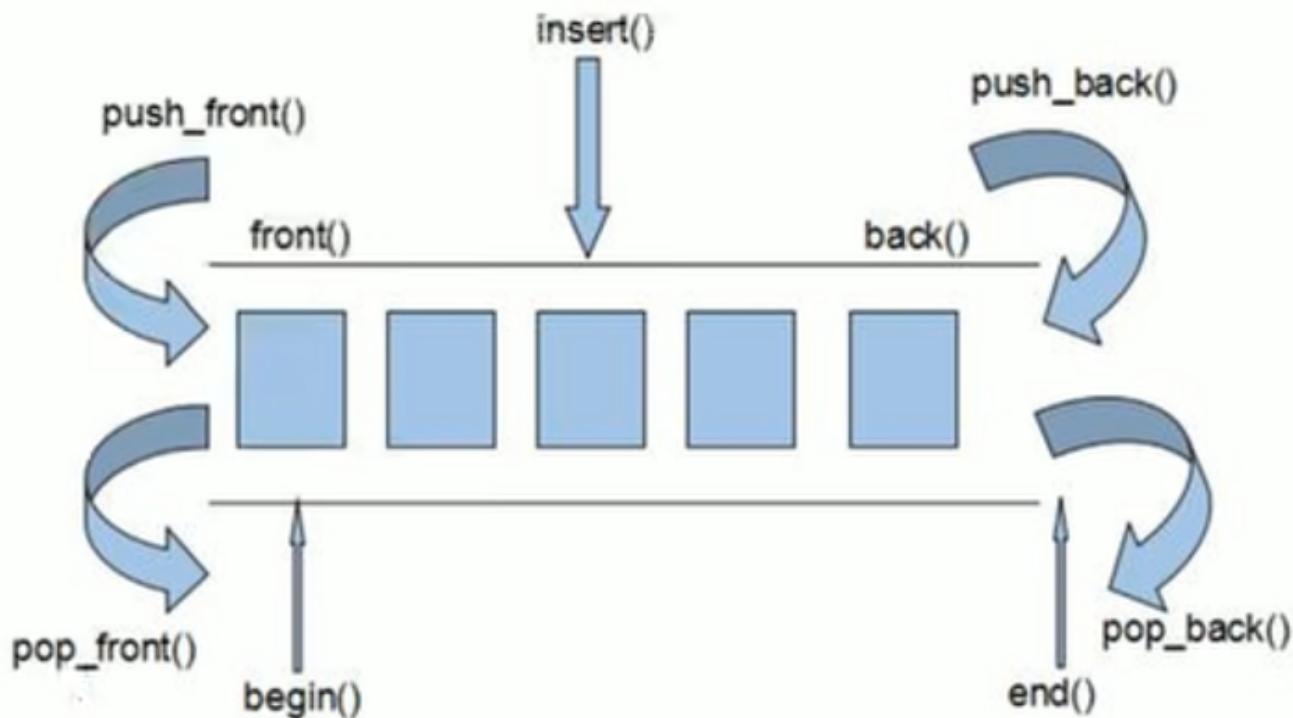
3.3.1 deque容器基本概念

功能:

- 双端数组，可以对头端进行插入删除操作

deque与vector区别:

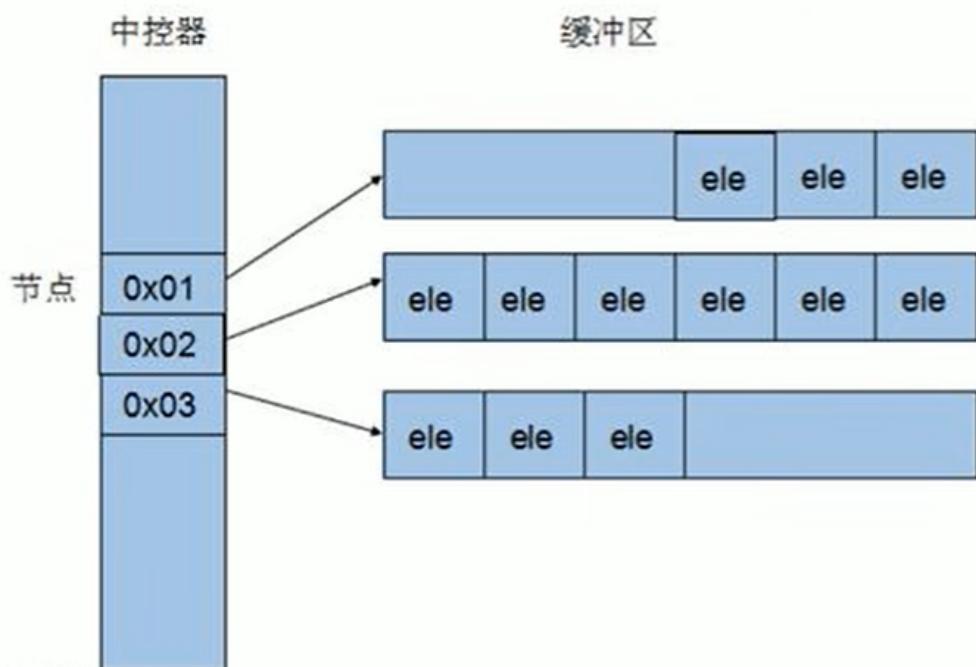
- vector对于头部的插入删除效率低，数据量越大，效率越低
- deque相对而言，对头部的插入删除速度会比vector快
- vector访问元素时的速度会比deque快，这和两者内部实现有关



deque内部工作原理:

deque内部有个**中控器**, 维护每段缓冲区中的内容, 缓冲区中存放真实**数据**

中控器维护的是每个缓冲区的地址, 使得使用deque时像一片连续的内存空间



- deque容器的迭代器也是支持随机访问的

3.3.2 deque构造函数

功能描述：

- deque容器构造

函数原型：

- `deque<T> deqT;` //默认构造形式
- `deque(beg, end);` //构造函数将[beg, end)区间中的元素拷贝给本身。
- `deque(n, elem);` //构造函数将n个elem拷贝给本身。
- `deque(const deque &deq);` //拷贝构造函数

3.3.3 deque赋值操作

功能描述：

- 给deque容器进行赋值

函数原型：

- `deque& operator=(const deque &deq);` //重载等号操作符
- `assign(beg, end);` //将[beg, end)区间中的数据拷贝赋值给本身。
- `assign(n, elem);` //将n个elem拷贝赋值给本身。

3.3.4 deque大小操作

功能描述：

- 对deque容器的大小进行操作

函数原型：

- `deque.empty();` //判断容器是否为空
- `deque.size();` //返回容器中元素的个数
- `deque.resize(num);` //重新指定容器的长度为num,若容器变长，则以默认值填充新位置。
//如果容器变短，则末尾超出容器长度的元素被删除。
- `deque.resize(num, elem);` //重新指定容器的长度为num,若容器变长，则以elem值填充新位置。
[//如果容器变短，则末尾超出容器长度的元素被删除。

3.3.5 deque插入和删除

功能描述：

- 向deque容器中插入和删除数据

函数原型：

两端插入操作：

- `push_back(elem);` //在容器尾部添加一个数据
- `push_front(elem);` //在容器头部插入一个数据
- `pop_back();` //删除容器最后一个数据
- `pop_front();` //删除容器第一个数据

指定位置操作：

- `insert(pos, elem);` //在pos位置插入一个elem元素的拷贝，返回新数据的位置。
- `insert(pos, n, elem);` //在pos位置插入n个elem数据，无返回值。
- `insert(pos, beg, end);` //在pos位置插入[beg,end)区间的数据，无返回值。
- `clear();` //清空容器的所有数据
- `erase(beg, end);` //删除[beg,end)区间的数据，返回下一个数据的位置。
- `erase(pos);` //删除pos位置的数据，返回下一个数据的位置。

总结：

- 插入和删除提供的位置是迭代器！
- 尾插 - `pushback`
- 尾删 - `pop_back`
- 头插 - `push_front`
- 头删 - `pop_front`

3.3.6 deque数据存取

功能描述:

- 对deque中的数据的存取操作

函数原型:

- `at(int idx);` //返回索引idx所指的数据
- `operator[];` //返回索引idx所指的数据
- `front();` //返回容器中第一个数据元素
- `back();` //返回容器中最后一个数据元素

总结:

- 除了用迭代器获取deque容器中元素，[]和at也可以
- front返回容器第一个元素
- back返回容器最后一个元素

3.3.7 deque排序

功能描述:

- 利用算法实现对deque容器进行排序

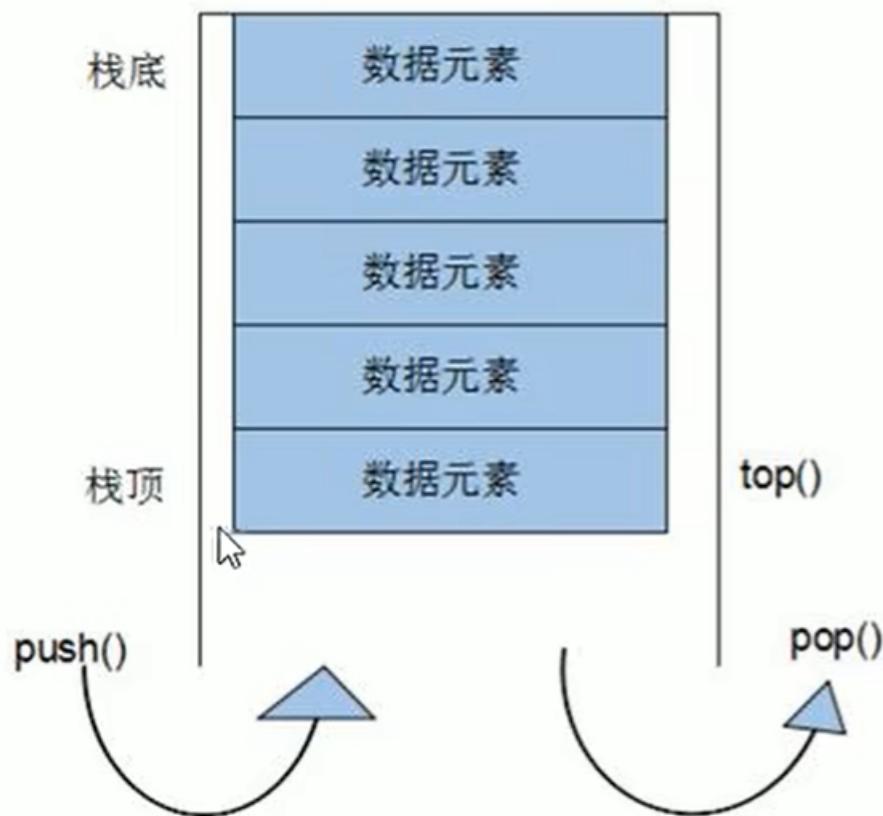
算法:

- `sort(iterator beg, iterator end)` //对beg和end区间内元素进行排序

3.4 stack容器

3.4.1 stack基本概念

概念：stack是一种先进后出(First In Last Out,FILO)的数据结构，它只有一个出口



栈中只有顶端的元素才可以被外界使用，因此栈不允许有遍历行为

3.4.2 stack常用接口

功能描述：栈容器常用的对外接口

构造函数：

- `stack<T> stk;` //stack采用模板类实现，stack对象的默认构造形式
- `stack(const stack &stk);` //拷贝构造函数

赋值操作：

- `stack& operator=(const stack &stk);` //重载等号操作符

数据存取：

- `push(elem);` //向栈顶添加元素
- `pop();` //从栈顶移除第一个元素
- `top();` //返回栈顶元素

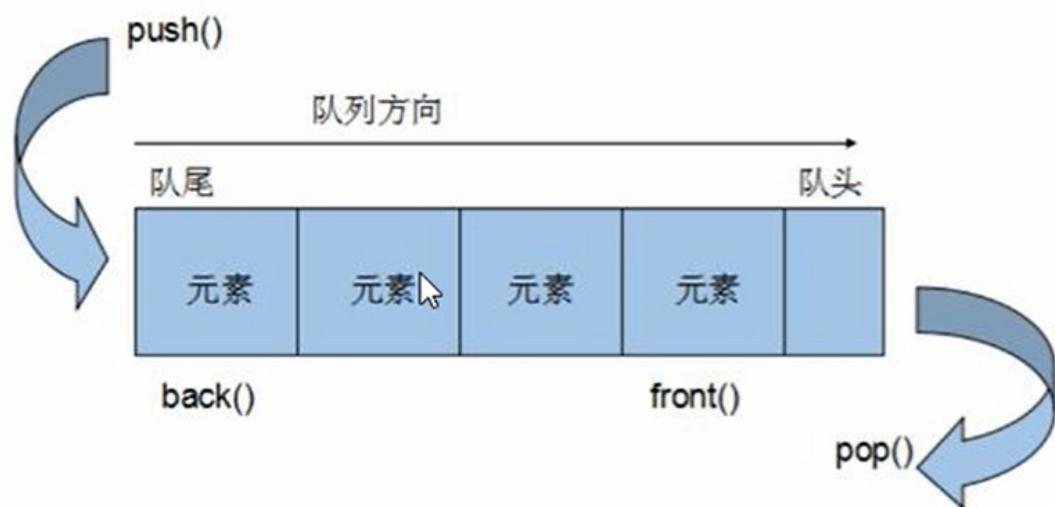
大小操作：

- `empty();` //判断堆栈是否为空
- `size();` //返回栈的大小

3.5 queue容器

3.5.1 queue基本概念

概念：Queue是一种先进先出(First In First Out,FIFO)的数据结构，它有两个出口



队列容器允许从一端新增元素，从另一端移除元素

队列中只有队头和队尾才可以被外界使用，因此队列不允许有遍历行为

队列中进数据称为 --- 入队 `push`

队列中出数据称为 --- 出队 `pop`

3.5.2 queue常用接口

功能描述：栈容器常用的对外接口

构造函数：

- `queue<T> que;` //queue采用模板类实现，que对象的默认构造形式
- `queue(const queue &que);` //拷贝构造函数

赋值操作：

- `queue& operator=(const queue &que);` //重载等号操作符

数据存取：

- `push(elem);` //往队尾添加元素
- `pop();` //从队头移除第一个元素
- `back();` //返回最后一个元素
- `front();` //返回第一个元素

大小操作：

- `empty();` //判断堆栈是否为空
- `size();` //返回栈的大小

3.6 list容器

3.6.1 list基本概念

功能：将数据进行链式存储

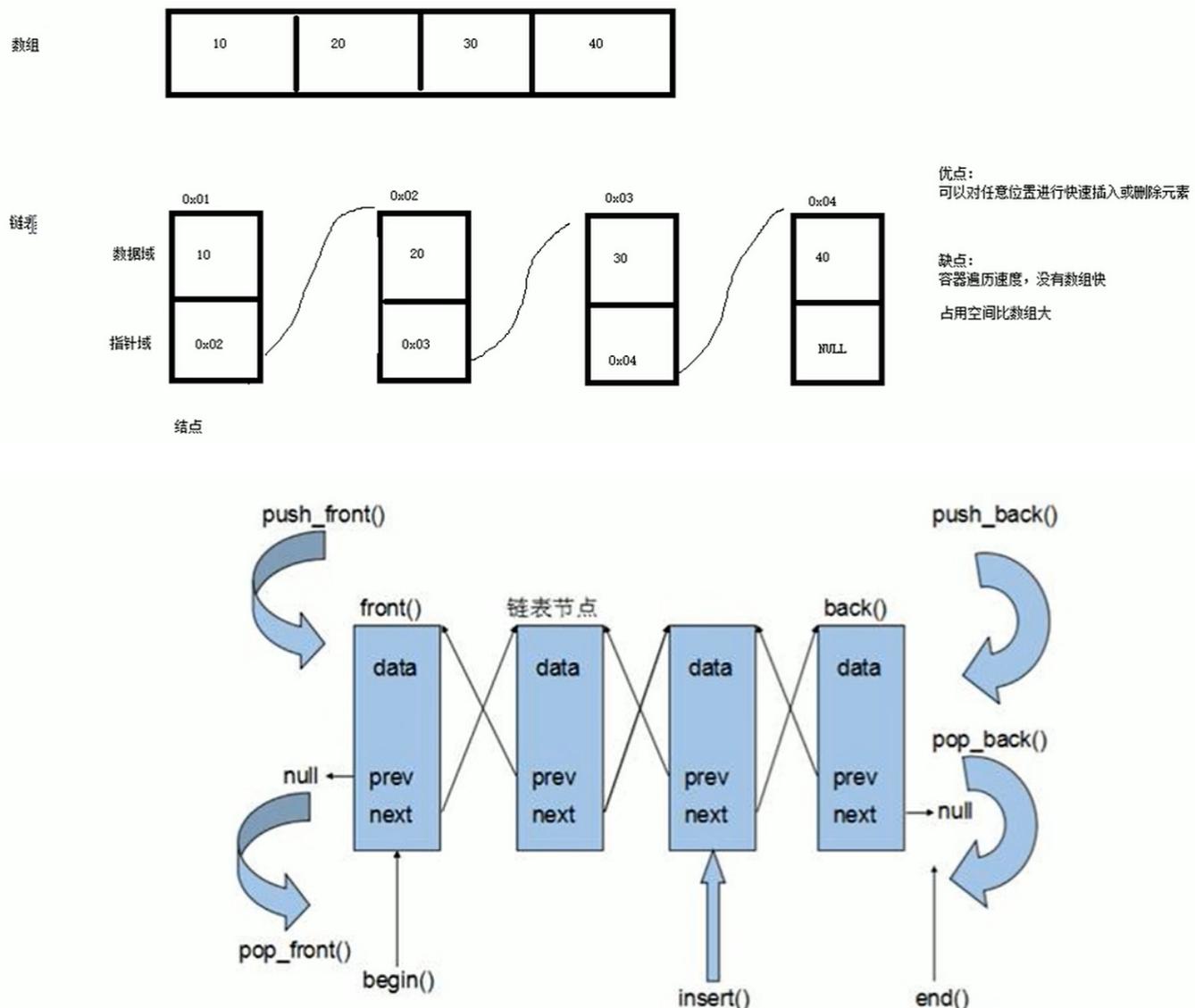
链表（list）是一种物理存储单元上非连续的存储结构，数据元素的逻辑顺序是通过链表中的指针链接实现的

I

链表的组成：链表由一系列结点组成

结点的组成：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域

STL中的链表是一个双向循环链表



由于链表的存储方式并不是连续的内存空间，因此链表list中的迭代器只支持前移和后移，属于双向迭代器

list的优点：

- 采用动态存储分配，不会造成内存浪费和溢出
- 链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素

list的缺点：

- 链表灵活，但是空间(指针域) 和 时间 (遍历) 额外耗费较大

List有一个重要的性质，插入操作和删除操作都不会造成原有list迭代器的失效，这在vector是不成立的。

总结：STL中**List**和**vector**是两个最常被使用的容器，各有优缺点

3.6.2 list构造函数

- 创建list容器

函数原型:

- `list<T> lst;` //list采用采用模板类实现,对象的默认构造形式:
- `list(beg, end);` //构造函数将[beg, end)区间中的元素拷贝给本身。
- `list(n, elem);` //构造函数将n个elem拷贝给本身。
- `list(const list &lst);` //拷贝构造函数。

3.6.3 list赋值和交换

功能描述:

- 给list容器进行赋值, 以及交换list容器

函数原型:

- `assign(beg, end);` //将[beg, end)区间中的数据拷贝赋值给本身。
- `assign(n, elem);` //将n个elem拷贝赋值给本身。
- `list& operator=(const list &lst);` //重载等号操作符
- `swap(lst);` //将lst与本身的元素互换。

3.6.4 list大小操作

功能描述:

- 对list容器的大小进行操作

函数原型:

- `size();` //返回容器中元素的个数
- `empty();` //判断容器是否为空
- `resize(num);` //重新指定容器的长度为num, 若容器变长, 则以默认值填充新位置。
//如果容器变短, 则末尾超出容器长度的元素被删除。
- `resize(num, elem);` //重新指定容器的长度为num, 若容器变长, 则以elem值填充新位置。
//如果容器变短, 则末尾超出容器长度的元素被删除。

3.6.5 list插入和删除

功能描述：

- 对list容器进行数据的插入和删除

函数原型：

- push_back(elem); //在容器尾部加入一个元素
- pop_back(); //删除容器中最后一个元素
- push_front(elem); //在容器开头插入一个元素
- pop_front(); //从容器开头移除第一个元素
- insert(pos, elem); //在pos位置插elem元素的拷贝，返回新数据的位置。
- insert(pos, n, elem); //在pos位置插入n个elem数据，无返回值。
- insert(pos, beg, end); //在pos位置插入[beg, end)区间的数据，无返回值。
- clear(); //移除容器的所有数据
- erase(beg, end); //删除[beg, end)区间的数据，返回下一个数据的位置。
- erase(pos); //删除pos位置的数据，返回下一个数据的位置。
- remove(elem); //删除容器中所有与elem值匹配的元素。

总结：

- 尾插 --- `push_back`
- 尾删 --- `pop_back`
- 头插 --- `push_front`
- 头删 --- `pop_front`
- 插入 --- `insert`
- 删除 --- `erase` [
- 移除 --- `remove`
- 清空 --- `clear`

3.6.6 list数据存取

功能描述：

- 对list容器中数据进行存取

函数原型：

- `front();` //返回第一个元素。
- `back();` //返回最后一个元素。

总结：

- list容器中不可以通过[]或者at方式访问数据
- 返回第一个元素 --- `front`
- 返回最后一个元素 --- `back`

3.6.6 list反转和排序

功能描述：

- 将容器中的元素反转，以及将容器中的数据进行排序

函数原型：

- `reverse();` //反转链表
- `sort();` //链表排序

3.7 set/multiset容器

3.7.1 set基本概念

简介：

- 所有元素都会在插入时自动被排序

本质：

- set/multiset属于关联式容器，底层结构是用二叉树实现。

set和multiset区别：

- set不允许容器中有重复的元素
- multiset允许容器中有重复的元素

3.7.2 set构造和赋值

功能描述：创建set容器以及赋值

构造：

- `set<T> st;` //默认构造函数：
- `set(const set &st);` //拷贝构造函数

赋值：

- `set& operator=(const set &st);` //重载等号操作符

3.8 map/multimap容器

3.8.1 map

简介:

- map中所有元素都是pair
- pair中第一个元素为key (键值) , 起到索引作用, 第二个元素为value (实值)
- 所有元素都会根据元素的键值自动排序

本质:

- map/multimap属于关联式容器, 底层结构是用二叉树实现。

优点:

- 可以根据key值快速找到value值

map和multimap区别:

- map不允许容器中有重复key值元素
- multimap允许容器中有重复key值元素

功能描述:

- 对map容器进行构造和赋值操作

函数原型:

构造:

- `map<T1, T2> mp;` //map默认构造函数;
- `map(const map &mp);` //拷贝构造函数

赋值:

- `map& operator=(const map &mp);` //重载等号操作符

功能描述：

- 统计map容器大小以及交换map容器

函数原型：

- `size();` //返回容器中元素的数目
- `empty();` //判断容器是否为空
- `swap(st);` //交换两个集合容器

功能描述：

- map容器进行插入数据和删除数据

函数原型：

- `insert(elem);` //在容器中插入元素。
- `clear();` //清除所有元素
- `erase(pos);` //删除pos迭代器所指的元素，返回下一个元素的迭代器。
- `erase(beg, end);` //删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
- `erase(key);` //删除容器中值为key的元素。

功能描述:

- 对map容器进行查找数据以及统计数据

函数原型:

- `find(key);` [] //查找key是否存在,若存在, 返回该键的元素的迭代器; 若不存在, 返回set.end();
- `count(key);` //统计key的元素个数

学习目标:

- map容器默认排序规则为 按照key值进行 从小到大排序, 掌握如何改变排序规则

主要技术点:

- 利用仿函数, 可以改变排序规则

STL函数对象

4.1 函数对象

4.1 函数对象

4.1.1 函数对象概念

概念：

- 重载函数调用操作符的类，其对象常称为**函数对象**
- 函数对象使用重载的()时，行为类似函数调用，也叫**仿函数**

本质：

函数对象(仿函数)是一个类，不是一个函数

4.1.2 函数对象使用

特点：

- 函数对象在使用时，可以像普通函数那样调用，可以有参数，可以有返回值
- 函数对象超出普通函数的概念，函数对象可以有自己的状态
- 函数对象可以作为参数传递

4.2 谓词

概念：

- 返回bool类型的仿函数称为**谓词**
- 如果operator()接受一个参数，那么叫做**一元谓词**
- 如果operator()接受两个参数，那么叫做**二元谓词**

4.3 内建函数对象

4.3.1 内建函数对象意义

概念：

- STL内建了一些函数对象

分类：

- 算术仿函数
- 关系仿函数
- 逻辑仿函数

用法：

- 这些仿函数所产生的对象，用法和一般函数完全相同
- 使用内建函数对象，需要引入头文件 `#include<functional>`

4.3.2 算术仿函数

功能描述：

- 实现四则运算
- 其中`negate`是一元运算，其他都是二元运算

仿函数原型：

- `template<class T> T plus<T>` //加法仿函数
- `template<class T> T minus<T>` //减法仿函数
- `template<class T> T multiplies<T>` //乘法仿函数
- `template<class T> T divides<T>` //除法仿函数
- `template<class T> T modulus<T>` //取模仿函数
- `template<class T> T negate<T>` //取反仿函数

4.3.3 关系仿函数

功能描述:

- 实现关系对比

仿函数原型:

- `template<class T> bool equal_to<T>` //等于
- `template<class T> bool not_equal_to<T>` //不等于
- `template<class T> bool greater<T>` //大于
- `template<class T> bool greater_equal<T>` //大于等于
- `template<class T> bool less<T>` //小于
- `template<class T> bool less_equal<T>` //小于等于

4.3.4 逻辑仿函数

功能描述:

- 实现逻辑运算

函数原型:

- `template<class T> bool logical_and<T>` //逻辑与
- `template<class T> bool logical_or<T>` //逻辑或
- `template<class T> bool logical_not<T>` //逻辑非

5 STL常用算法

概述:

- 算法主要是由头文件 `<algorithm>` `<functional>` `<numeric>` 组成。
- `<algorithm>` 是所有STL头文件中最大的一个，范围涉及到比较、交换、查找、遍历操作、复制、修改等等
- `<numeric>` 体积很小，只包括几个在序列上面进行简单数学运算的模板函数
- `<functional>` 定义了一些模板类，用以声明函数对象。

5.1 常用遍历算法

5.1 常用遍历算法

学习目标:

- 掌握常用的遍历算法

算法简介:

- for_each //遍历容器
- transform //搬运容器到另一个容器中

5.1.1 for_each

功能描述:

- 实现遍历容器

函数原型:

- for_each(iterator beg, iterator end, Tfunc);
// 遍历算法 遍历容器元素
// beg 开始迭代器
// end 结束迭代器
// _func 函数或者函数对象

5.1.2 transform

功能描述：

- 搬运容器到另一个容器中

函数原型：

- `transform(iterator beg1, iterator end1, iterator beg2, _func);`

//beg1 源容器开始迭代器

//end1 源容器结束迭代器

//beg2 目标容器开始迭代器

//_func 函数或者函数对象

```
vector<int>vTarget; //目标容器
```

```
vTarget.resize(v.size()); // 目标容器需要提前开辟空间
```

```
transform(v.begin(), v.end(), vTarget.begin(), TransForm());  
+
```

```
for_each(vTarget.begin(), vTarget.end(), MyPrint());
```

5.2 常用的查找算法

算法简介：

- `find` //查找元素
- `find_if` //按条件查找元素
- `adjacent_find` //查找相邻重复元素
- `binary_search` //二分查找
- `count` //统计元素个数
- `count_if` //按条件统计元素个数

5.2.1 find

功能描述：

- 查找指定元素，找到返回指定元素的迭代器，找不到返回结束迭代器`end()`

函数原型：

- `find(iterator beg, iterator end, value);`
// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
// beg 开始迭代器
// end 结束迭代器
// value 查找的元素

5.2.2 find_if

功能描述：

- 按条件查找元素

函数原型：

- ```
find_if(iterator beg, iterator end, _Pred);
```

// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置  
// beg 开始迭代器  
// end 结束迭代器  
// \_Pred 函数或者谓词（返回bool类型的仿函数）

## 5.3 常用的排序算法

## 算法简介：

- `sort()` // 对容器内元素进行排序
- `random_shuffle()` // 洗牌 指定范围内的元素随机调整次序
- `merge()` // 容器元素合并，并存储到另一容器中
- `reverse()` // 反转指定范围的元素

### 5.3.2 random\_shuffle

#### 功能描述：

- 洗牌 指定范围内的元素随机调整次序

#### 函数原型：

- `random_shuffle(iterator beg, iterator end);`  
    // 指定范围内的元素随机调整次序  
    // beg 开始迭代器  
    // end 结束迭代器

### 5.3.3 merge

功能描述:

- 两个容器元素合并，并存储到另一容器中

函数原型:

- `merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`  
// 容器元素合并，并存储到另一容器中  
// 注意: 两个容器必须是有序的  
// beg1 容器1开始迭代器  
// end1 容器1结束迭代器  
// beg2 容器2开始迭代器  
// end2 容器2结束迭代器  
// dest 目标容器开始迭代器

### 5.3.4 reverse

功能描述:

- 将容器内元素进行反转

I

函数原型:

- `reverse(iterator beg, iterator end);`  
// 反转指定范围的元素  
// beg 开始迭代器  
// end 结束迭代器

## 5.4 常用的拷贝和替换算法

## 算法简介：

- `copy` [ ] // 容器内指定范围的元素拷贝到另一容器中
- `replace` [ ] // 将容器内指定范围的旧元素修改为新元素
- `replace_if` [ ] // 容器内指定范围满足条件的元素替换为新元素
- `swap` [ ] // 互换两个容器的元素

### 5.4.1 copy

#### 功能描述：

- 容器内指定范围的元素拷贝到另一容器中

#### 函数原型：

- `copy(iterator beg, iterator end, iterator dest);`  
// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置  
// beg 开始迭代器  
// end 结束迭代器  
// dest 目标起始迭代器

## 5.4.2 replace

功能描述:

I

- 将容器内指定范围的旧元素修改为新元素

函数原型:

- `replace(iterator beg, iterator end, oldvalue, newvalue);`  
// 将区间内旧元素替换成新元素  
// beg 开始迭代器  
// end 结束迭代器  
// oldvalue 旧元素  
// newvalue 新元素

## 5.4.3 replace\_if

功能描述:

- 将区间内满足条件的元素，替换成指定元素

函数原型:

- `replace_if(iterator beg, iterator end, _pred, newvalue);`  
// 按条件替换元素，满足条件的替换成指定元素  
// beg 开始迭代器  
// end 结束迭代器  
// \_pred 谓词  
// newvalue 替换的新元素

#### 5.4.4 swap

[

功能描述：

- 互换两个容器的元素

函数原型：

- ```
swap(container c1, container c2);
```

// 互换两个容器的元素
// c1容器1
// c2容器2

5.5 常用的算数生成法

5.5 常用算术生成算法

学习目标：

- 掌握常用的算术生成算法

注意：

- 算术生成算法属于小型算法，使用时包含的头文件为 `#include <numeric>`

算法简介：

- `accumulate` // 计算容器元素累计总和
- `fill` // 向容器中添加元素

5.5.1 accumulate

功能描述：

- 计算区间内 容器元素累计总和

函数原型：

- `accumulate(iterator beg, iterator end, value);`
// 计算容器元素累计总和
// beg 开始迭代器
// end 结束迭代器
// value 起始值

5.5.2 fill

功能描述：

- 向容器中填充指定的元素

函数原型：

- `fill(iterator beg, iterator end, value);`

// 向容器中填充元素

// beg 开始迭代器

// end 结束迭代器

// value 填充的值

5.6 常用集合

5.6 常用集合算法

学习目标:

- 掌握常用的集合算法

算法简介:

- `set_intersection` // 求两个容器的交集
- `set_union` // 求两个容器的并集
- `set_difference` // 求两个容器的差集

5.6.1 set_intersection

功能描述:

- 求两个容器的交集

函数原型:

```
• set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);  
// 求两个集合的交集  
// 注意:两个集合必须是有序序列  
// beg1 容器1开始迭代器  
// end1 容器1结束迭代器  
// beg2 容器2开始迭代器  
// end2 容器2结束迭代器  
// dest 目标容器开始迭代器
```

5.6.2 set_union

I

功能描述:

- 求两个集合的并集

函数原型:

- ```
set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

  
// 求两个集合的并集  
// 注意:两个集合必须是有序序列  
// beg1 容器1开始迭代器  
// end1 容器1结束迭代器  
// beg2 容器2开始迭代器  
// end2 容器2结束迭代器  
// dest 目标容器开始迭代器

## 5.6.3 set\_difference

I

功能描述:

- 求两个集合的差集

函数原型:

- ```
set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```


// 求两个集合的差集
// 注意:两个集合必须是有序序列
// beg1 容器1开始迭代器
// end1 容器1结束迭代器
// beg2 容器2开始迭代器
// end2 容器2结束迭代器
// dest 目标容器开始迭代器

类型转换

1. 显示类型转换和隐式类型转换

- 当等号两边的类型不同的时候、形参与实参类型不匹配的时候、返回值类型与接收返回值类型不一致时，就需要发生类型转化。
- 而类型转换又分为隐式类型转换和显示类型转换。

```
int main() {  
    // 隐式类型转换  
    int Ival = 1;  
    double Dval = Ival;
```

```

cout << "Dval: " << Dval << endl;

//显式类型转换
int* p = &Ival;
int pi = p; //error
int pi = (int)p;
cout << "pi: " << pi << endl;
system("pause");
return(0);
}

```

- 隐式类型转换是编译器在编译阶段自动进行，能转就转，不能转就编译失败。而显示类型转换就要我们自己处理。

2.C++的四种强制类型转换

- 上面的两种类型转换是C语言风格的，存在一些缺点。隐式类型转换会造成精度的丢失。而显示类型转换则会导致转换不清晰（不知道谁转化过来）。所以C++提供了规范的四种类型转换。

2.1 static_cast

- 相似转化
- 如果想要进行相似类型的转换，编译器隐式执行的任何类型转换都可用。但是如果是**两个不相关的类型**就不能转换。

```

int main() {
    int i = 0;
    double d = static_cast<double>(i);
    int* p = &i;
    int pi = static_cast<int>(p); //error
    system("pause");
    return(0);
}

```

2.2 reinterpret_cast

- 不同类型转化
- 上面我们用指针类型转化成整型出现错误，而这种不同类型的转换要用**reinterpret_cast**。

```

int main() {
    int i = 0;
    double d = static_cast<double>(i);
    int* p = &i;
    int pi = static_cast<int>(p); //error

    int pi = reinterpret_cast<int>(p);
    system("pause");
}

```

```
    return(0);
}
```

2.3 const_cast

- 去除const属性
- 使用`const_cast`的主要目的是为了去除一个const变量的const，方便赋值。

```
int main() {
    //volatile const int a = 2;
    const int a = 2;
    int* p = const_cast<int*>(&a);
    *p = 3;
    cout << a << endl; //2
    cout << *p << endl; //3
    system("pause");
    return(0);
}
```

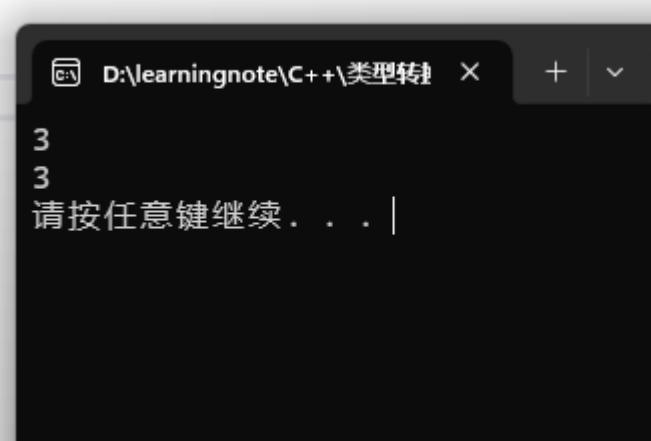
这里的结果需要注意一下：

```
cout << a << endl; //2          2
cout << *p << endl; //3          3
system("pause");                请按任意键继续. . . |
```

```
cout << a << endl; //2
cout << *p << endl; //3
```

这里是因为编译器把这个变量放到了寄存器中，我们修改的是内存中的数据，不影响寄存器，我们可以加上`volatile`关键字（每次都去内存中取）来看看：

```
int main() {
    volatile const int a = 2;
    //const int a = 2;
    int* p = const_cast<int*>(&a);
    *p = 3;
    cout << a << endl; //2
    cout << *p << endl; //3
    system("pause");
    return(0);
}
```



2.4 dynamic_cast

向上转型： 子类的指针（或引用） \rightarrow 父类的指针（或引用）。

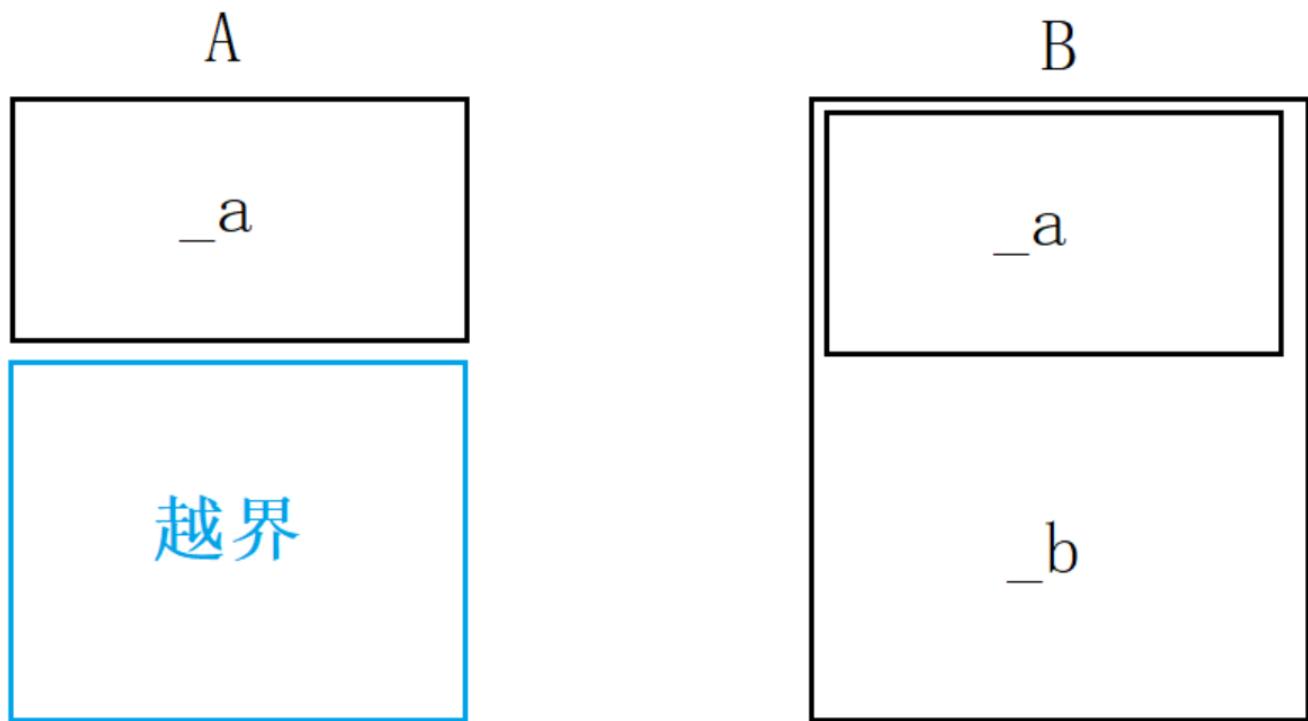
向下转型： 父类的指针（或引用） \rightarrow 子类的指针（或引用）。

- 其中，向上转型就是所说的切割/切片，是语法天然支持的，不需要进行转换，而向下转型是语法不支持的，需要进行强制类型转换。

向下转换

- dynamic_cast用于将一个父类对象的指针/引用转换为子类对象的指针或引用(动态转换)
- C++继承中讲过，子类对象赋值给父类 对象/指针/引用，这里有个形象的说法叫切片或者切割，寓意把派生类中父类那部分切来赋值过去。
- 但是如果我们将父类对象传递给子类，会不安全，因为父类转给子类会多开一份空间，可能会越界访问。

```
class A {  
public:  
    virtual void f() {};  
    int _a = 0;  
};  
  
class B : public::A {  
public:  
    int _b = 0;  
};  
  
void fun(A* pa) {  
    B* pb = (B*)pa;  
  
    pb->_a++;  
    pb->_b++;  
}  
  
int main() {  
    A a;  
    B b;  
    fun(&a);  
    fun(&b);  
    system("pause");  
    return(0);  
}
```



- 而加上`dynamic_cast`后如果转化失败就会返回空指针，让我们检查：

```

//void fun(A* pa) {
//    B* pb = (B*)pa;
//    ...
//    pb->_a++;
//    pb->_b++;
//}

void fun(A* pa) {
    B* pb = dynamic_cast<B*>(pa);
    pb->_a++; // Error: pb is nullptr
    pb->_b++;
}

void fun(A* pa) {
    B* pb = dynamic_cast<B*>(pa);
    cout << pb << endl;
    if (pb) {
        pb->_a++;
        pb->_b++;
    }
}

int main() {
    A a;
    B b;
    fun(&a);
    fun(&b);
    system("pause");
    return(0);
}

```

已引发异常
引发了异常: 读取访问权限冲突。
pb 是 nullptr.

询问 Copilot | 显示调用堆栈 | 复制详细信息 | 启动 Live Share 会话
异常设置
引发此异常类型时中断
从以下位置引发时除外:
□ 类型转换.exe
打开异常设置 | 编辑条件

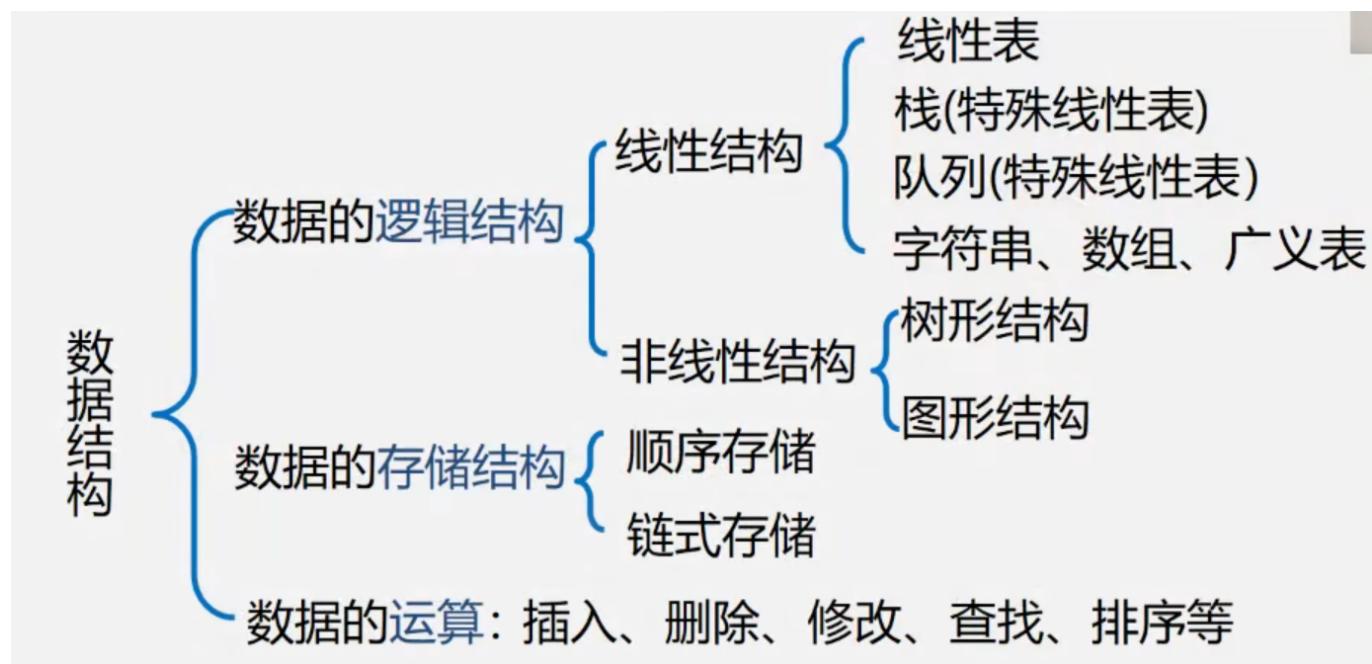
D:\learningnote\C++\类型转换 x +
0000000000000000
0000000F0FF8FC98
请按任意键继续. . . |

3. 总结

4种类型转换的应用场景

- static_cast用于相近类型的类型之间的转换，编译器隐式执行的任何类型转换都可用static_cast。
- reinterpret_cast用于两个不相关类型之间的转换。
- const_cast用于删除变量的const属性，方便赋值。
- dynamic_cast用于安全的将父类的指针（或引用）转换成子类的指针（或引用）。

数据结构思维导图



哈希表

hash又翻译为散列

1. 基本思想

- **基本思想**: 记录的存储位置与关键字之间存在对应关系

对应关系——hash函数

$$\text{Loc}(i) = H(\text{key}_i)$$



hash函数



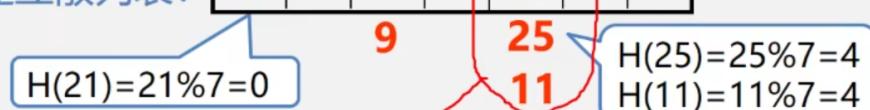
冲突: 不同的关键码映射到同一个散列地址

$\text{key}_1 \neq \text{key}_2$, 但是 $H(\text{key}_1) = H(\text{key}_2)$

例: 有6个元素的关键码分别为: (25, 21, 39, 9, 23, 11)。

- 选取关键码与元素位置间的函数为 $H(k) = k \bmod 7$,
- 地址编号从0-6。

通过散列函数对6个元素建立散列表:



同义词: 具有相同函数值的多个关键字

有冲突!

使用散列表要解决好两个问题:

1) 构造好的散列函数

- 所选函数尽可能简单, 以便提高转换速度;
- 所选函数对关键码计算出的地址, 应在散列地址集中致均匀分布, 以减少空间浪费。

2) 制定一个好的解决冲突的方案

查找时, 如果从散列函数计算出的地址中查不到关键码, 则应当依据解决冲突的规则, 有规律地查询其它相关单元。

2. 哈希函数构造方法

根据元素集合的特性构造

- **要求一：**n个数据原仅占用n个地址，虽然散列查找是以空间换时间，但仍希望散列的**地址空间尽量小**。
- **要求二：**无论用什么方法存储，目的都是尽量**均匀**地存放元素，以避免冲突。



1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

2.1. 直接定址法

直接定址法

$$\text{Hash(key)} = a \cdot \text{key} + b \quad (a, b \text{ 为常数})$$

优点：以关键码key的某个线性函数值为散列地址，不会产生冲突。

缺点：要占用连续地址空间，空间效率低。

例：{100, 300, 500, 700, 800, 900},

散列函数 $\text{Hash(key)} = \text{key}/100 \quad (a=1/100, b=0)$

0 1 2 3 4 5 6 7 8 9

	100		300		500		700	800	900
--	-----	--	-----	--	-----	--	-----	-----	-----

2.2. 除留余数法

除留余数法

$$\text{Hash(key)} = \text{key} \bmod p \quad (p \text{是一个整数})$$

关键：如何选取合适的p?

技巧：设表长为m，取 $p \leq m$ 且为质数

例： {15, 23, 27, 38, 53, 61, 70},

散列函数 $\text{Hash(key)} = \text{key} \bmod 7$

0 1 2 3 4 5 6

70	15	23	38	53	61	27
----	----	----	----	----	----	----

3. 处理哈希冲突的方法

处理冲突的方法

1. 开放定址法 (开地址法)

2. 链地址法 (拉链法)

3. 再散列法 (双散列函数法)

4. 建立一个公共溢出区

3.1. 开放定址法

基本思想：有冲突时就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将数据元素存入。

例如：除留余数法 $H_i = (\text{Hash}(\text{key}) + d_i) \bmod m$ d_i 为增量序列

常用方法： 线性探测法 d_i 为 1, 2, ... $m-1$ 线性序列

二次探测法 d_i 为 $1^2, -1^2, 2^2, -2^2, \dots, q^2$ 二次序列

伪随机探测法 d_i 为伪随机数序列

例如：

二次探测法

关键码集为 {47, 7, 29, 11, 16, 92, 22, 8, 3},

设： 散列函数为 $\text{Hash}(\text{key}) = \text{key} \bmod 11$

$$H_i = (\text{Hash}(\text{key}) + d_i) \bmod m$$

其中：m为散列表长度，m要求是某个 $4k+3$ 的质数；

d_i 为增量序列 $1^2, -1^2, 2^2, -2^2, \dots, q^2$

0	1	2	3	4	5	6	7	8	9	10
11	22	3	47	92	16		7	29	8	

Hash(3)=3, 散列地址冲突, 由
 $H_1 = (\text{Hash}(3) + 1^2) \bmod 11 = 4$,
 仍然冲突;
 $H_2 = (\text{Hash}(3) - 1^2) \bmod 11 = 2$,
 找到空的散列地址, 存入。

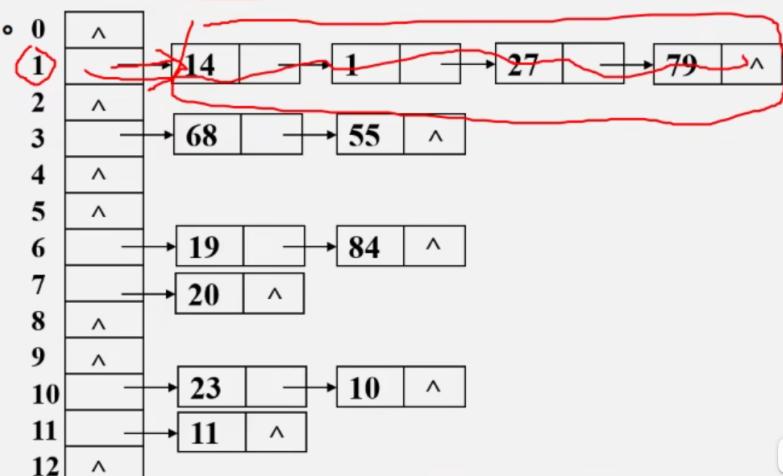
3.2. 链地址法 (拉链法)

基本思想：相同散列地址的记录链成一单链表

m个散列地址就设m个单链表，然后用一个数组将m个单链表的表头指针存储起来，形成一个动态的结构。

例如：一组关键字为
 $\{19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79\}$,

散列函数为
 $\text{Hash}(\text{key}) = \text{key} \bmod 13$



- 链地址法建立哈希表步骤

链地址法建立散列表步骤

- Step1: 取数据元素的关键字key，计算其散列函数值（地址）。若该地址对应的链表为空，则将该元素插入此链表；否则执行Step2解决冲突。
- Step2: 根据选择的冲突处理方法，计算关键字key的下一个存储地址。若该地址对应的链表为不为空，则利用链表的前插法或后插法将该元素插入此链表。

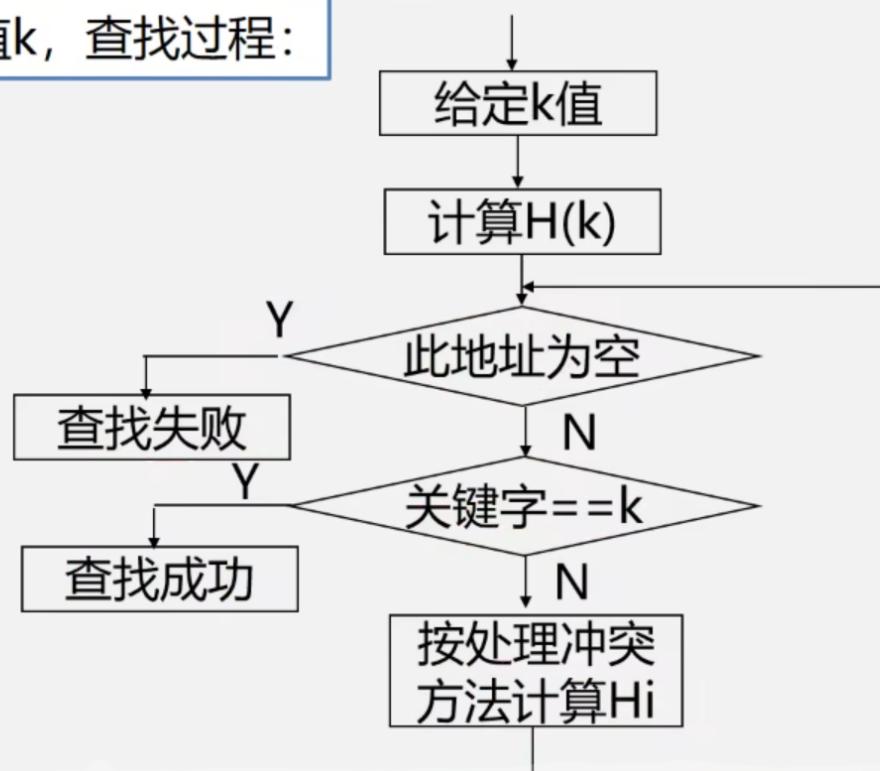
- 链地址法的优点

链地址法的优点：

- 非同义词不会冲突，无“聚集”现象
- 链表上结点空间动态申请，更适用于表长不确定的情况

4. 哈希表的查找

给定值查找值k，查找过程：



平均查找长度

使用平均查找长度ASL来衡量查找算法，ASL取决于

- 散列函数
- 处理冲突的方法
- 散列表的装填因子 α

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

α 越大，表中记录数越多，说明表装得越满，发生冲突的可能性就越大，查找时比较次数就越多。

散列表的查找效率分析

ASL与装填因子 α 有关！既不是严格的O(1)，也不是O(n)

$$ASL \approx 1 + \frac{\alpha}{2} \quad (\text{拉链法})$$

$$ASL \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha}\right) \quad (\text{线性探测法})$$

$$ASL \approx -\frac{1}{\alpha} \ln(1 - \alpha) \quad (\text{随机探测法})$$

- 哈希表具有很好的平均性能
- 链地址法优于开放地址法（1是查找效率，2是动态的表的长度可以变化）
- 除留余数法做哈希函数优于其他类型函数（除数 p 一般是小于等于哈希表大小 m 的最大质数）

常见的树

1. 平衡二叉树 (AVL树)

- 一颗平衡二叉树或者是空树，或者是具有下列性质的二叉排序树。

1. 左子树与右子树的高度差的绝对值小于等于1

2. 左子树和右子树也是平衡二叉排序树

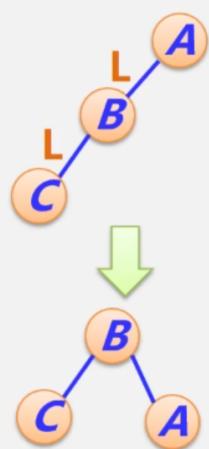


- 插入新节点失衡时

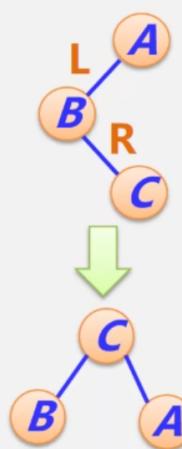
失衡二叉排序树的分析与调整

平衡调整的四种类型：

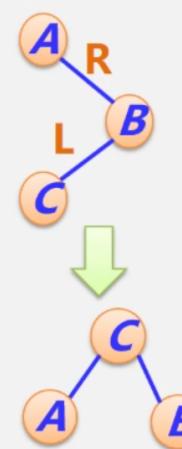
LL型



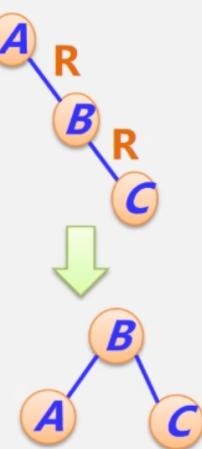
LR型



RL型



RR型



调整原则：1) 降低高度 2) 保持二叉排序树性质